«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Факультет программной инженерии и компьютерной техники

Отчет

По лабораторной работе №5

Вариант 1674

Студент:

*Ильин Н. С.*

*Р3110 поток 2.9*

Преподаватель:

*Бойко В.А.*

Санкт-Петербург, 2024 г.

**Оглавление**

[Задание: 3](#_Toc145408419)

[Выполнение работы: 3](#_Toc145408420)

[Выводы: 45](#_Toc145408421)

# **Задание:**

**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.Vector
* При запуске приложения коллекция должна автоматически заполняться значениями из файла.
* Имя файла должно передаваться программе с помощью: **переменная окружения**.
* Данные должны храниться в файле в формате json
* Чтение данных из файла необходимо реализовать с помощью класса java.util.Scanner
* Запись данных в файл необходимо реализовать с помощью класса java.io.FileOutputStream
* Все классы в программе должны быть задокументированы в формате javadoc.
* Программа должна корректно работать с неправильными данными (ошибки пользовательского ввода, отсутсвие прав доступа к файлу и т.п.).

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

* help : вывести справку по доступным командам
* info : вывести в стандартный поток вывода информацию о коллекции (тип, дата инициализации, количество элементов и т.д.)
* show : вывести в стандартный поток вывода все элементы коллекции в строковом представлении
* add {element} : добавить новый элемент в коллекцию
* update id {element} : обновить значение элемента коллекции, id которого равен заданному
* remove\_by\_id id : удалить элемент из коллекции по его id
* clear : очистить коллекцию
* save : сохранить коллекцию в файл
* execute\_script file\_name : считать и исполнить скрипт из указанного файла. В скрипте содержатся команды в таком же виде, в котором их вводит пользователь в интерактивном режиме.
* exit : завершить программу (без сохранения в файл)
* insert\_at index {element} : добавить новый элемент в заданную позицию
* add\_if\_max {element} : добавить новый элемент в коллекцию, если его значение превышает значение наибольшего элемента этой коллекции
* sort : отсортировать коллекцию в естественном порядке
* average\_of\_age : вывести среднее значение поля age для всех элементов коллекции
* group\_counting\_by\_creation\_date : сгруппировать элементы коллекции по значению поля creationDate, вывести количество элементов в каждой группе
* print\_unique\_color : вывести уникальные значения поля color всех элементов в коллекции

**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 Dragon {

private int id; //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.LocalDateTime creationDate; //Поле не может быть null, Значение этого поля должно генерироваться автоматически

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

private Boolean speaking; //Поле может быть null

private Color color; //Поле не может быть null

private DragonCharacter character; //Поле может быть null

private Person killer; //Поле может быть null

}

public class Coordinates {

private double x;

private Long y; //Поле не может быть null

}

public class Person {

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

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

private Float height; //Поле может быть null, Значение поля должно быть больше 0

private Long weight; //Поле может быть null, Значение поля должно быть больше 0

private Location location; //Поле может быть null

}

public class Location {

private Float x; //Поле не может быть null

private float y;

private Long z; //Поле не может быть null

}

public enum Color {

RED,

YELLOW,

ORANGE;

}

public enum DragonCharacter {

CUNNING,

WISE,

CHAOTIC\_EVIL;

}

**Отчёт по работе должен содержать:**

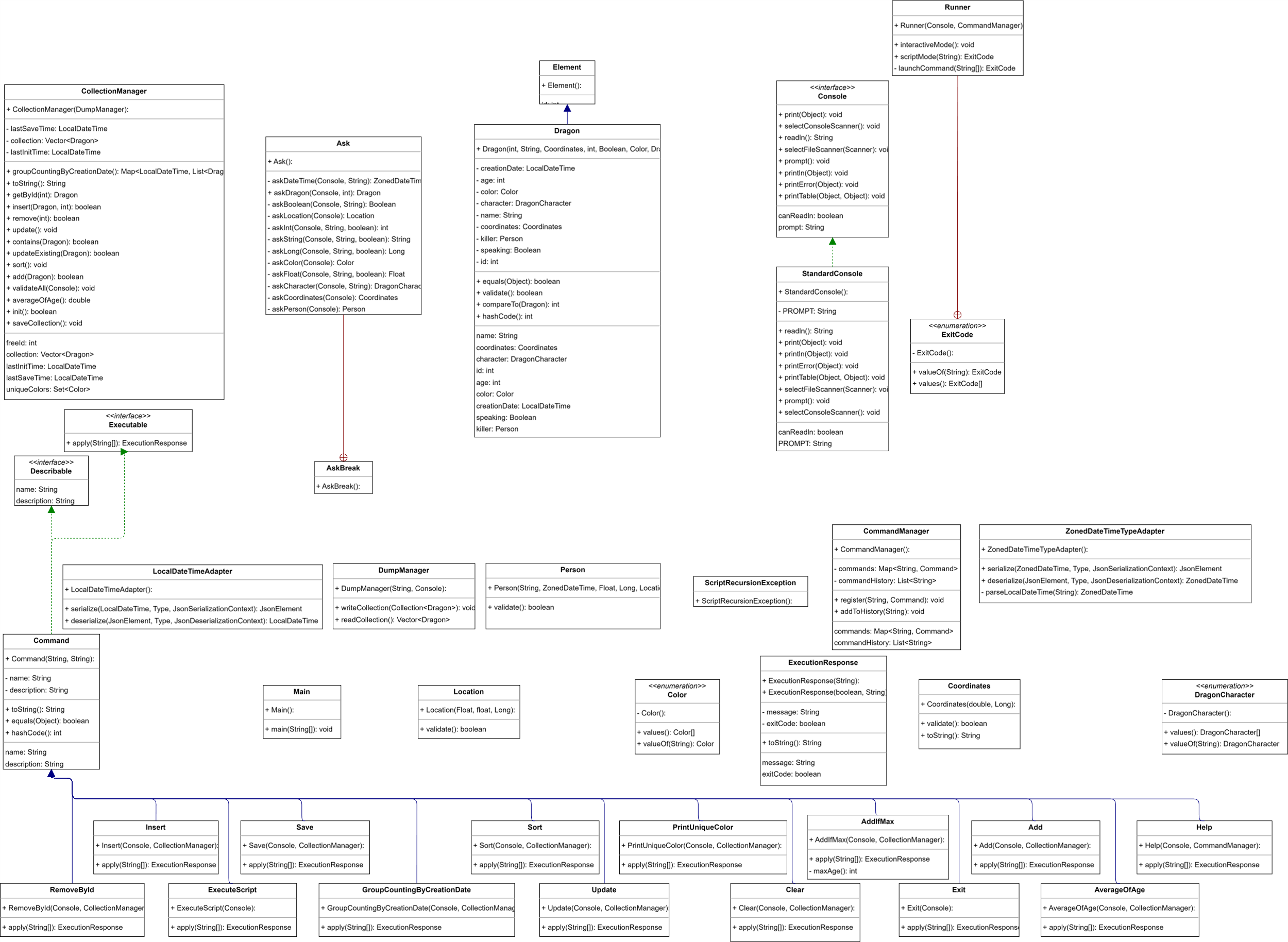
1. Текст задания.
2. Диаграмма классов разработанной программы.
3. Исходный код программы.
4. Выводы по работе.

**Вопросы к защите лабораторной работы:**

1. Коллекции. Сортировка элементов коллекции. Интерфейсы java.util.Comparable и java.util.Comparator.
2. Категории коллекций - списки, множества. Интерфейс java.util.Map и его реализации.
3. Параметризованные типы. Создание параметризуемых классов. Wildcard-параметры.
4. Классы-оболочки. Назначение, область применения, преимущества и недостатки. Автоупаковка и автораспаковка.
5. Потоки ввода-вывода в Java. Байтовые и символьные потоки. "Цепочки" потоков (Stream Chains).
6. Работа с файлами в Java. Класс java.io.File.
7. Пакет java.nio - назначение, основные классы и интерфейсы.
8. Утилита javadoc. Особенности автоматического документирования кода в Java.

# **Выполнение работы:**

**UML-диаграмма:**



**Github:**  
<https://github.com/MrTheFall/study/tree/master/2sem/lab5>

**Исходный код:**

# main/java/org/lab5/Main.java

**package** **org.lab5**;

**import** **org.lab5.commands.\***;

**import** **org.lab5.managers.CollectionManager**;

**import** **org.lab5.managers.CommandManager**;

**import** **org.lab5.managers.DumpManager**;

**import** **org.lab5.utils.Runner**;

**import** **org.lab5.utils.console.StandardConsole**;

**public** **class** **Main** {

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

**var** console = **new** StandardConsole();

String filename = System.getenv("FILENAME");

**if** (filename == **null** || filename.isEmpty()) {

System.out.println("Задайте имя загружаемого файла с помощью переменной среды FILENAME");

System.exit(**1**);

}

System.out.println(filename);

**var** dumpManager = **new** DumpManager(filename, console);

**var** collectionManager = **new** CollectionManager(dumpManager);

**if** (!collectionManager.init()) { System.exit(**1**); }

collectionManager.validateAll(console);

**var** commandManager = **new** CommandManager() {{

register("help", **new** Help(console, **this**));

register("add", **new** Add(console, collectionManager));

register("update", **new** Update(console, collectionManager));

register("remove\_by\_id", **new** RemoveById(console, collectionManager));

register("clear", **new** Clear(console, collectionManager));

register("save", **new** Save(console, collectionManager));

register("execute\_script", **new** ExecuteScript(console));

register("exit", **new** Exit(console));

register("add\_if\_max", **new** AddIfMax(console, collectionManager));

register("insert", **new** Insert(console, collectionManager));

register("sort", **new** Sort(console, collectionManager));

register("average\_of\_age", **new** AverageOfAge(console, collectionManager));

register("group\_counting\_by\_creation\_date", **new** GroupCountingByCreationDate(console, collectionManager));

register("print\_unique\_color", **new** PrintUniqueColor(console, collectionManager));

}};

**new** Runner(console, commandManager).interactiveMode();

}

}

# main/java/org/lab5/commands/Add.java

**package** **org.lab5.commands**;

**import** **org.lab5.managers.CollectionManager**;

**import** **org.lab5.models.Dragon**;

**import** **org.lab5.utils.ExecutionResponse**;

**import** **org.lab5.utils.console.Console**;

**import** **org.lab5.utils.Ask**;

/\*\*

\* Команда 'add'. Добавляет новый элемент в коллекцию.

\*/

**public** **class** **Add** **extends** Command {

**private** **final** Console console;

**private** **final** CollectionManager collectionManager;

/\*\*

\* @param console

\* @param collectionManager

\*/

**public** **Add**(Console console, CollectionManager collectionManager) {

**super**("add {element}", "добавить новый элемент в коллекцию");

**this**.console = console;

**this**.collectionManager = collectionManager;

}

/\*\*

\* Выполняет команду

\* @return Успешность выполнения команды и сообщение об успешности.

\*/

**@Override**

**public** ExecutionResponse **apply**(String[] arguments) {

**try** {

**if** (!arguments[**1**].isEmpty()) {

console.printError("Неправильное количество аргументов!\nИспользование: '" + getName() + "'");

**return** **new** ExecutionResponse(**false**,

"Неправильное количество аргументов!\nИспользование: '" + getName() + "'");

}

console.println("\* Создание нового Дракона:");

Dragon dragon = Ask.askDragon(console, collectionManager.getFreeId());

**if** (dragon != **null** && dragon.validate()) {

collectionManager.add(dragon);

console.println("Дракон успешно добавлен!");

**return** **new** ExecutionResponse("Дракон успешно добавлен!");

} **else**

**return** **new** ExecutionResponse(**false**, "Поля дракона не валидны! Дракон не создан!");

} **catch** (Ask.AskBreak e) {

**return** **new** ExecutionResponse(**false**, "Отмена...");

}

}

}

# main/java/org/lab5/commands/AddIfMax.java

**package** **org.lab5.commands**;

**import** **org.lab5.managers.CollectionManager**;

**import** **org.lab5.models.Dragon**;

**import** **org.lab5.utils.Ask**;

**import** **org.lab5.utils.ExecutionResponse**;

**import** **org.lab5.utils.console.Console**;

/\*\*

\* Команда 'add\_if\_max'. Добавляет новый элемент в коллекцию, если его значение превышает значение наибольшего элемента этой коллекции.

\*/

**public** **class** **AddIfMax** **extends** Command {

**private** **final** Console console;

**private** **final** CollectionManager collectionManager;

/\*\*

\* @param console

\* @param collectionManager

\*/

**public** **AddIfMax**(Console console, CollectionManager collectionManager) {

**super**("add\_if\_max {element}", "добавить новый элемент в коллекцию, если его значение превышает значение наибольшего элемента этой коллекции");

**this**.console = console;

**this**.collectionManager = collectionManager;

}

/\*\*

\* Выполняет команду

\* @return Успешность выполнения команды и сообщение об успешности.

\*/

**@Override**

**public** ExecutionResponse **apply**(String[] arguments) {

**try** {

**if** (!arguments[**1**].isEmpty()) {

console.printError("Неправильное количество аргументов!\nИспользование: '" + getName() + "'");

**return** **new** ExecutionResponse(**false**,

"Неправильное количество аргументов!\nИспользование: '" + getName() + "'");

}

console.println("\* Создание нового продукта (add\_if\_max):");

Dragon dragon = Ask.askDragon(console, collectionManager.getFreeId());

**int** maxAge = maxAge();

**if** (maxAge == -**1** || dragon.getAge() > maxAge) {

collectionManager.add(dragon);

console.println("Дракон успешно добавлен!");

} **else** {

console.println("Дракон не добавлен, его возрасть не превышает возраст самого старого дракона (" + dragon.getAge() + " < " + maxAge + ")");

}

**return** **new** ExecutionResponse(**true**, "Дракон успешно добавлен!");

} **catch** (Ask.AskBreak e) {

**return** **new** ExecutionResponse(**false**, "Отмена...");

}

}

**private** **int** **maxAge**() {

**return** collectionManager.getCollection().stream()

.map(Dragon::getAge)

.mapToInt(Integer::intValue)

.max()

.orElse(-**1**);

}

}

# main/java/org/lab5/commands/AverageOfAge.java

**package** **org.lab5.commands**;

**import** **org.lab5.managers.CollectionManager**;

**import** **org.lab5.utils.ExecutionResponse**;

**import** **org.lab5.utils.console.Console**;

/\*\*

\* Команда 'average\_of\_age'. Вычисляет средний возраст всех драконов в коллекции.

\*/

**public** **class** **AverageOfAge** **extends** Command {

**private** **final** Console console;

**private** **final** CollectionManager collectionManager;

/\*\*

\* @param console

\* @param collectionManager

\*/

**public** **AverageOfAge**(Console console, CollectionManager collectionManager) {

**super**("average\_of\_age", "вычислить средний возраст всех драконов в коллекции");

**this**.console = console;

**this**.collectionManager = collectionManager;

}

/\*\*

\* Выполняет команду.

\* @return Успешность выполнения команды и сообщение об успешности.

\*/

**@Override**

**public** ExecutionResponse **apply**(String[] arguments) {

**try** {

// Вызываем метод averageOfAge у collectionManager и получаем средний возраст

**double** averageAge = collectionManager.averageOfAge();

**if** (Double.isNaN(averageAge)) { // Проверяем, что результат не NaN, т.е. в коллекции есть элементы

console.printError("Коллекция пуста, не удалось вычислить средний возраст.");

**return** **new** ExecutionResponse(**false**, "Коллекция пуста, не удалось вычислить средний возраст.");

}

// Выводим результат

console.println("Средний возраст драконов в коллекции: " + averageAge);

**return** **new** ExecutionResponse(**true**, "Средний возраст драконов: " + averageAge);

} **catch** (Exception e) {

console.printError("Произошла ошибка во время вычисления среднего возраста");

**return** **new** ExecutionResponse(**false**, "Произошла ошибка во время вычисления среднего возраста");

}

}

}

# main/java/org/lab5/commands/Clear.java

**package** **org.lab5.commands**;

**import** **org.lab5.managers.CollectionManager**;

**import** **org.lab5.models.Dragon**;

**import** **org.lab5.utils.ExecutionResponse**;

**import** **org.lab5.utils.console.Console**;

/\*\*

\* Команда 'clear'. Очищает коллекцию.

\*/

**public** **class** **Clear** **extends** Command {

**private** **final** Console console;

**private** **final** CollectionManager collectionManager;

**public** **Clear**(Console console, CollectionManager collectionManager) {

**super**("clear", "очистить коллекцию");

**this**.console = console;

**this**.collectionManager = collectionManager;

}

/\*\*

\* Выполняет команду

\* @return Успешность выполнения команды.

\*/

**@Override**

**public** ExecutionResponse **apply**(String[] arguments) {

**if** (!arguments[**1**].isEmpty()) {

console.println("Неправильное количество аргументов!");

console.println("Использование: '" + getName() + "'");

**return** **new** ExecutionResponse(**false**, "Неправильное количество аргументов!");

}

**while** (!collectionManager.getCollection().isEmpty()) {

**int** lastElementIndex = collectionManager.getCollection().size() - **1**;

Dragon dragon = collectionManager.getCollection().remove(lastElementIndex);

collectionManager.remove(dragon.getId());

}

collectionManager.update();

console.println("Коллекция очищена!");

**return** **new** ExecutionResponse(**true**, "Коллекция очищена!");

}

}

# main/java/org/lab5/commands/Command.java

**package** **org.lab5.commands**;

/\*\*

\* Абстрактная команда с именем и описанием

\*/

**public** **abstract** **class** **Command** **implements** Describable, Executable {

**private** **final** String name;

**private** **final** String description;

/\*\*

\* @param name

\* @param description

\*/

**public** **Command**(String name, String description) {

**this**.name = name;

**this**.description = description;

}

/\*\*

\* @return Название и использование команды.

\*/

**public** String **getName**() {

**return** name;

}

/\*\*

\* @return Описание команды.

\*/

**public** String **getDescription**() {

**return** description;

}

**@Override**

**public** **boolean** **equals**(Object obj) {

**if** (**this** == obj) **return** **true**;

**if** (obj == **null** || getClass() != obj.getClass()) **return** **false**;

Command command = (Command) obj;

**return** name.equals(command.name) && description.equals(command.description);

}

**@Override**

**public** **int** **hashCode**() {

**return** name.hashCode() + description.hashCode();

}

**@Override**

**public** String **toString**() {

**return** "Command{" +

"name='" + name + '\'' +

", description='" + description + '\'' +

'}';

}

}

# main/java/org/lab5/commands/Describable.java

**package** **org.lab5.commands**;

/\*\*

\* Интерфейс объектов, имеющих описание.

\*/

**public** **interface** **Describable** {

/\*\*

\* Получить имя.

\* @return имя

\*/

String **getName**();

/\*\*

\* Получить описание.

\* @return описание

\*/

String **getDescription**();

}

# main/java/org/lab5/commands/Executable.java

**package** **org.lab5.commands**;

**import** **org.lab5.utils.ExecutionResponse**;

/\*\*

\* Интерфейс выполняемых объектов.

\*/

**public** **interface** **Executable** {

/\*\*

\* Выполняет что-либо.

\* @param arguments Аргумент для выполнения

\* @return результат выполнения

\*/

ExecutionResponse **apply**(String[] arguments);

}

# main/java/org/lab5/commands/ExecuteScript.java

**package** **org.lab5.commands**;

**import** **org.lab5.utils.ExecutionResponse**;

**import** **org.lab5.utils.console.Console**;

/\*\*

\* Команда 'execute\_script'. Выполнить скрипт из файла.

\* @author maxbarsukov

\*/

**public** **class** **ExecuteScript** **extends** Command {

**private** **final** Console console;

**public** **ExecuteScript**(Console console) {

**super**("execute\_script <file\_name>", "исполнить скрипт из указанного файла");

**this**.console = console;

}

/\*\*

\* Выполняет команду

\* @return Успешность выполнения команды.

\*/

**@Override**

**public** ExecutionResponse **apply**(String[] arguments) {

**if** (arguments[**1**].isEmpty()) {

console.println("Использование: '" + getName() + "'");

**return** **new** ExecutionResponse(**true**, "Некорректное использование...");

}

console.println("Выполнение скрипта '" + arguments[**1**] + "'...");

**return** **new** ExecutionResponse(**true**, "Выполнение скрипта...");

}

}

# main/java/org/lab5/commands/Exit.java

**package** **org.lab5.commands**;

**import** **org.lab5.utils.ExecutionResponse**;

**import** **org.lab5.utils.console.Console**;

/\*\*

\* Команда 'exit'. Завершает программу без сохранения в файл.

\*/

**public** **class** **Exit** **extends** Command {

**private** **final** Console console;

/\*\*

\* @param console

\*/

**public** **Exit**(Console console) {

**super**("exit", "завершить программу (без сохранения в файл)");

**this**.console = console;

}

/\*\*

\* Выполняет команду

\* @return Успешность выполнения команды.

\*/

**@Override**

**public** ExecutionResponse **apply**(String[] arguments) {

**if** (!arguments[**1**].isEmpty()) {

console.printError("Неправильное количество аргументов!");

console.printError("Использование: '" + getName() + "'");

**return** **new** ExecutionResponse(**false**, "Неправильное количество аргументов!");

}

console.println("Завершение выполнения...");

**return** **new** ExecutionResponse(**true**, "Завершение выполнения...");

}

}

# main/java/org/lab5/commands/GroupCountingByCreationDate.java

**package** **org.lab5.commands**;

**import** **org.lab5.managers.CollectionManager**;

**import** **org.lab5.models.Dragon**;

**import** **org.lab5.utils.ExecutionResponse**;

**import** **org.lab5.utils.console.Console**;

**import** **java.time.LocalDateTime**;

**import** **java.util.List**;

**import** **java.util.Map**;

/\*\*

\* Команда 'group\_counting\_by\_creation\_date'. Группирует элементы коллекции по дате создания.

\*/

**public** **class** **GroupCountingByCreationDate** **extends** Command {

**private** **final** Console console;

**private** **final** CollectionManager collectionManager;

/\*\*

\* @param console

\* @param collectionManager

\*/

**public** **GroupCountingByCreationDate**(Console console, CollectionManager collectionManager) {

**super**("group\_counting\_by\_creation\_date", "группировать элементы коллекции по дате создания");

**this**.console = console;

**this**.collectionManager = collectionManager;

}

/\*\*

\* Выполняет команду

\* @return Успешность выполнения команды.

\*/

**@Override**

**public** ExecutionResponse **apply**(String[] arguments) {

Map<LocalDateTime, List<Dragon>> groups = collectionManager.groupCountingByCreationDate();

**if** (groups.isEmpty()) {

console.println("Коллекция пуста.");

**return** **new** ExecutionResponse(**true**, "Коллекция пуста.");

}

groups.forEach((date, dragons) -> {

console.println("Дата создания: " + date + " | Количество драконов: " + dragons.size());

});

**return** **new** ExecutionResponse(**true**, "Элементы коллекции успешно сгруппированы по дате создания.");

}

}

# main/java/org/lab5/commands/Help.java

**package** **org.lab5.commands**;

**import** **org.lab5.managers.CommandManager**;

**import** **org.lab5.utils.ExecutionResponse**;

**import** **org.lab5.utils.console.Console**;

/\*\*

\* Команда 'help'. Выводит справку по доступным командам

\*/

**public** **class** **Help** **extends** Command {

**private** **final** Console console;

**private** **final** CommandManager commandManager;

/\*\*

\* @param console

\* @param commandManager

\*/

**public** **Help**(Console console, CommandManager commandManager) {

**super**("help", "вывести справку по доступным командам");

**this**.console = console;

**this**.commandManager = commandManager;

}

/\*\*

\* Выполняет команду

\* @return Успешность выполнения команды.

\*/

**@Override**

**public** ExecutionResponse **apply**(String[] arguments) {

**if** (!arguments[**1**].isEmpty()) {

console.println("Использование: '" + getName() + "'");

**return** **new** ExecutionResponse("Выведена справка по команде"+ getName());

}

commandManager.getCommands().values().forEach(command -> {

console.printTable(command.getName(), command.getDescription());

});

**return** **new** ExecutionResponse("Справка выведена.");

}

}

# main/java/org/lab5/commands/Insert.java

**package** **org.lab5.commands**;

**import** **org.lab5.managers.CollectionManager**;

**import** **org.lab5.models.Dragon**;

**import** **org.lab5.utils.ExecutionResponse**;

**import** **org.lab5.utils.console.Console**;

**import** **org.lab5.utils.Ask**;

/\*\*

\* Команда 'insert'. Вставляет новый элемент в заданное место коллекции.

\*/

**public** **class** **Insert** **extends** Command {

**private** **final** Console console;

**private** **final** CollectionManager collectionManager;

/\*\*

\* @param console

\* @param collectionManager

\*/

**public** **Insert**(Console console, CollectionManager collectionManager) {

**super**("insert <ID> {element}", "вставить новый элемент в заданное место коллекции");

**this**.console = console;

**this**.collectionManager = collectionManager;

}

/\*\*

\* Выполняет команду.

\* @return Успешность выполнения команды и сообщение об успешности.

\*/

**@Override**

**public** ExecutionResponse **apply**(String[] arguments) {

**try** {

**if** (arguments.length < **2** || arguments[**1**].isEmpty()) {

console.printError("Укажите ID для вставки элемента!\nИспользование: '" + getName() + "'");

**return** **new** ExecutionResponse(**false**,

"Укажите ID для вставки элемента!\nИспользование: '" + getName() + "'");

}

**int** insertIndex = -**1**;

**try** {

insertIndex = Integer.parseInt(arguments[**1**].trim());

} **catch** (NumberFormatException e) {

console.printError("ID не распознан");

**return** **new** ExecutionResponse(**false**, "Отмена...");

}

console.println("\* Вставка элемента по индексу " + insertIndex + ":");

Dragon dragon = Ask.askDragon(console, collectionManager.getFreeId());

**if** (dragon != **null** && dragon.validate()) {

console.println(collectionManager.insert(dragon, insertIndex));

console.println("Дракон по индексу " + insertIndex + " успешно добавлен!");

**return** **new** ExecutionResponse(**true**, "Дракон по индексу " + insertIndex + " успешно добавлен!");

} **else** {

console.printError("Поля дракона не валидны! Дракон не добавлен.");

**return** **new** ExecutionResponse(**false**, "Поля дракона не валидны! Дракон не добавлен.");

}

} **catch** (Ask.AskBreak e) {

**return** **new** ExecutionResponse(**false**, "Отмена...");

}

}

}

# main/java/org/lab5/commands/PrintUniqueColor.java

**package** **org.lab5.commands**;

**import** **org.lab5.managers.CollectionManager**;

**import** **org.lab5.models.Color**;

**import** **org.lab5.utils.ExecutionResponse**;

**import** **org.lab5.utils.console.Console**;

**import** **java.util.Set**;

/\*\*

\* Команда 'print\_unique\_color'. Выводит уникальные значения поля color всех элементов в коллекции.

\*/

**public** **class** **PrintUniqueColor** **extends** Command {

**private** **final** Console console;

**private** **final** CollectionManager collectionManager;

/\*\*

\* @param console

\* @param collectionManager

\*/

**public** **PrintUniqueColor**(Console console, CollectionManager collectionManager) {

**super**("print\_unique\_color", "вывести уникальные значения поля color всех элементов в коллекции");

**this**.console = console;

**this**.collectionManager = collectionManager;

}

/\*\*

\* Выполняет команду

\* @return Успешность выполнения команды.

\*/

**@Override**

**public** ExecutionResponse **apply**(String[] arguments) {

Set<Color> uniqueColors = collectionManager.getUniqueColors();

**if** (uniqueColors.isEmpty()) {

console.printError("В коллекции нет элементов, поэтому уникальные цвета отсутствуют.");

**return** **new** ExecutionResponse(**true**, "Уникальные цвета отсутствуют.");

}

console.println("Уникальные цвета драконов в коллекции:");

uniqueColors.forEach(color -> {

console.println("Цвет: " + color);

});

**return** **new** ExecutionResponse(**true**, "Уникальные цвета успешно выведены.");

}

}

# main/java/org/lab5/commands/RemoveById.java

**package** **org.lab5.commands**;

**import** **org.lab5.managers.CollectionManager**;

**import** **org.lab5.utils.ExecutionResponse**;

**import** **org.lab5.utils.console.Console**;

/\*\*

\* Команда 'remove\_by\_id'. Удаляет элемент из коллекции по ID.

\*/

**public** **class** **RemoveById** **extends** Command {

**private** **final** Console console;

**private** **final** CollectionManager collectionManager;

/\*\*

\* @param console

\* @param collectionManager

\*/

**public** **RemoveById**(Console console, CollectionManager collectionManager) {

**super**("remove\_by\_id <ID>", "удалить элемент из коллекции по ID");

**this**.console = console;

**this**.collectionManager = collectionManager;

}

/\*\*

\* Выполняет команду

\* @return Успешность выполнения команды.

\*/

**@Override**

**public** ExecutionResponse **apply**(String[] arguments) {

**if** (arguments[**1**].isEmpty()) {

console.println("Неправильное количество аргументов!");

console.println("Использование: '" + getName() + "'");

**return** **new** ExecutionResponse(**false**, "Неправильное количество аргументов!");

}

**int** id = -**1**;

**try** {

id = Integer.parseInt(arguments[**1**].trim());

} **catch** (NumberFormatException e) {

console.println("ID не распознан");

**return** **new** ExecutionResponse(**false**, "ID не распознан");

}

**if** (collectionManager.getById(id) == **null** || !collectionManager.getCollection().contains(collectionManager.getById(id))) {

console.println("не существующий ID");

**return** **new** ExecutionResponse(**false**, "не существующий ID");

}

collectionManager.remove(id);

collectionManager.update();

console.println("Дракон успешно удалён!");

**return** **new** ExecutionResponse(**true**, "Дракон успешно удалён!");

}

}

# main/java/org/lab5/commands/Save.java

**package** **org.lab5.commands**;

**import** **org.lab5.managers.CollectionManager**;

**import** **org.lab5.utils.ExecutionResponse**;

**import** **org.lab5.utils.console.Console**;

/\*\*

\* Команда 'save'. Сохраняет коллекцию в файл.

\*/

**public** **class** **Save** **extends** Command {

**private** **final** Console console;

**private** **final** CollectionManager collectionManager;

/\*\*

\* @param console

\* @param collectionManager

\*/

**public** **Save**(Console console, CollectionManager collectionManager) {

**super**("save", "сохранить коллекцию в файл");

**this**.console = console;

**this**.collectionManager = collectionManager;

}

/\*\*

\* Выполняет команду

\* @return Успешность выполнения команды.

\*/

**@Override**

**public** ExecutionResponse **apply**(String[] arguments) {

**if** (!arguments[**1**].isEmpty()) {

console.println("Неправильное количество аргументов!");

console.println("Использование: '" + getName() + "'");

**return** **new** ExecutionResponse(**false**, "Неправильное количество аргументов!");

}

collectionManager.saveCollection();

**return** **new** ExecutionResponse(**true**, "Коллекция сохранена!");

}

}

# main/java/org/lab5/commands/Sort.java

**package** **org.lab5.commands**;

**import** **org.lab5.managers.CollectionManager**;

**import** **org.lab5.utils.ExecutionResponse**;

**import** **org.lab5.utils.console.Console**;

/\*\*

\* Команда 'sort'. Сортирует коллекцию в естественном порядке.

\*/

**public** **class** **Sort** **extends** Command {

**private** **final** Console console;

**private** **final** CollectionManager collectionManager;

/\*\*

\* @param console

\* @param collectionManager

\*/

**public** **Sort**(Console console, CollectionManager collectionManager) {

**super**("sort", "сортировка коллекции в естественном порядке");

**this**.console = console;

**this**.collectionManager = collectionManager;

}

/\*\*

\* Выполняет команду сортировки.

\* @return Успешность выполнения команды и сообщение об успешности.

\*/

**@Override**

**public** ExecutionResponse **apply**(String[] arguments) {

**try** {

console.println("\* Сортировка коллекции:");

collectionManager.sort();

console.println("Коллекция успешно отсортирована!");

**return** **new** ExecutionResponse(**true**, "Коллекция успешно отсортирована!");

} **catch** (Exception e) {

console.printError("Возникла ошибка при попытке сортировки коллекции: " + e.getMessage());

**return** **new** ExecutionResponse(**false**, "Возникла ошибка при попытке сортировки коллекции: " + e.getMessage());

}

}

}

# main/java/org/lab5/commands/Update.java

**package** **org.lab5.commands**;

**import** **org.lab5.managers.CollectionManager**;

**import** **org.lab5.utils.Ask**;

**import** **org.lab5.utils.ExecutionResponse**;

**import** **org.lab5.utils.console.Console**;

/\*\*

\* Команда 'update'. Обновляет значение элемента коллекции по ID.

\*/

**public** **class** **Update** **extends** Command {

**private** **final** Console console;

**private** **final** CollectionManager collectionManager;

/\*\*

\* @param console

\* @param collectionManager

\*/

**public** **Update**(Console console, CollectionManager collectionManager) {

**super**("update <ID> {element}", "обновить значение элемента коллекции по ID");

**this**.console = console;

**this**.collectionManager = collectionManager;

}

/\*\*

\* Выполняет команду

\* @return Успешность выполнения команды.

\*/

**@Override**

**public** ExecutionResponse **apply**(String[] arguments) {

**try** {

**if** (arguments[**1**].isEmpty()) {

console.println("Неправильное количество аргументов!");

console.println("Использование: '" + getName() + "'");

**return** **new** ExecutionResponse(**false**, "Неправильное количество аргументов!");

}

**int** id = -**1**;

**try** {

id = Integer.parseInt(arguments[**1**].trim());

} **catch** (NumberFormatException e) {

console.println("ID не распознан");

**return** **new** ExecutionResponse(**false**, "Отмена...");

}

**var** old = collectionManager.getById(id);

**if** (old == **null** || !collectionManager.getCollection().contains(old)) {

console.println("не существующий ID");

**return** **new** ExecutionResponse(**false**, "не существующий ID!");

}

console.println("\* Создание нового Дракона:");

**var** d = Ask.askDragon(console, old.getId());

**if** (d != **null** && d.validate()) {

collectionManager.remove(old.getId());

collectionManager.add(d);

collectionManager.update();

**return** **new** ExecutionResponse(**true**, "Обновление дракона!");

} **else** {

console.println("Поля Дракона не валидны! Дракон не создан!");

**return** **new** ExecutionResponse(**false**, "Поля Дракона не валидны! Дракон не обновлен!");

}

} **catch** (Ask.AskBreak e) {

console.println("Отмена...");

**return** **new** ExecutionResponse(**false**, "Отмена...");

}

}

}

# main/java/org/lab5/exceptions/ScriptRecursionException.java

**package** **org.lab5.exceptions**;

/\*\*

\* Исключение. Выбрасывается, если скрипт попадает в бесконечную рекурсию.

\*/

**public** **class** **ScriptRecursionException** **extends** Exception {}

# main/java/org/lab5/managers/CollectionManager.java

**package** **org.lab5.managers**;

**import** **org.lab5.models.Color**;

**import** **org.lab5.models.Dragon**;

**import** **org.lab5.utils.console.Console**;

**import** **java.time.LocalDate**;

**import** **java.time.LocalDateTime**;

**import** **java.util.\***;

**import** **java.util.concurrent.atomic.AtomicInteger**;

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

/\*\*

\* Менеджер коллекции.

\*/

**public** **class** **CollectionManager** {

**private** **int** currentId = **1**;

**private** Vector<Dragon> collection = **new** Vector<>();

**private** LocalDateTime lastInitTime;

**private** LocalDateTime lastSaveTime;

**private** **final** DumpManager dumpManager;

/\*\*

\* @param dumpManager

\*/

**public** **CollectionManager**(DumpManager dumpManager) {

**this**.lastInitTime = **null**;

**this**.lastSaveTime = **null**;

**this**.dumpManager = dumpManager;

}

/\*\*

\* @return последнее время инициализации в формате LocalDateTime

\*/

**public** LocalDateTime **getLastInitTime**() {

**return** lastInitTime;

}

/\*\*

\* @return последнее время сохранения в формате LocalDateTime

\*/

**public** LocalDateTime **getLastSaveTime**() {

**return** lastSaveTime;

}

/\*\*

\* @return коллекция драконов.

\*/

**public** Vector<Dragon> **getCollection**() {

**return** collection;

}

/\*\*

\* @return Элемент коллекции по заданному ID

\*/

**public** Dragon **getById**(**int** id) {

**return** collection.stream().filter(dragon -> dragon.getId() == id).findFirst().orElse(**null**);

}

/\*\*

\* Проверяет, содержится ли переданный экземпляр дракона в коллекции. Проверка производится по id.

\* @return Boolean, есть ли дракон в коллекции.

\*/

**public** **boolean** **contains**(Dragon dragon) {

**return** dragon == **null** || getById(dragon.getId()) != **null**;

}

/\*\*

\* Получает незанятый id.

\* @return Свободный ID.

\*/

**public** **int** **getFreeId**() {

**while** (getById(currentId) != **null**)

**if** (++currentId < **0**)

currentId = **1**;

**return** currentId;

}

/\*\*

\* Добавляет дракона в коллекцию на определенную позицию.

\* @param dragon дракон для добавления.

\* @param insertIndex позиция для добавления.

\* @return true, если дракон был успешно добавлен, иначе - false.

\*/

**public** **boolean** **insert**(Dragon dragon, **int** insertIndex) {

**if** (contains(dragon)) **return** **false**;

collection.insertElementAt(dragon, insertIndex);

//update();

**return** **true**;

}

/\*\*

\* Добавляет дракона в коллекцию.

\* @param dragon дракон для добавления.

\* @return true, если дракон был успешно добавлен, иначе - false.

\*/

**public** **boolean** **add**(Dragon dragon) {

**if** (contains(dragon)) **return** **false**;

collection.add(dragon);

update();

**return** **true**;

}

/\*\*

\* Обновляет существующего дракона в коллекции.

\* @param dragon дракон с обновленными данными.

\* @return true, если дракон был успешно обновлен, иначе - false.

\*/

**public** **boolean** **updateExisting**(Dragon dragon) {

**if** (!contains(dragon)) **return** **false**;

collection.removeIf(existingDragon -> existingDragon.getId() == dragon.getId());

collection.add(dragon);

update();

**return** **true**;

}

/\*\*

\* Удаляет дракона из коллекции по его идентификатору.

\* @param id идентификатор дракона для удаления.

\* @return true, если дракон был успешно удален, иначе - false.

\*/

**public** **boolean** **remove**(**int** id) {

Dragon dragonToRemove = getById(id);

**if** (dragonToRemove == **null**) **return** **false**;

collection.remove(dragonToRemove);

update();

**return** **true**;

}

/\*\*

\* Обновляет и сортирует коллекцию.

\*/

**public** **void** **update**() {

**this**.sort(); // Dragon реализует Comparable<Dragon>

}

/\*\*

\* Сортирует коллекцию.

\*/

**public** **void** **sort**() { Collections.sort(collection); }

/\*\*

\* Расчитывает средний возраст драконов в коллекции.

\* @return средний возраст драконов.

\*/

**public** **double** **averageOfAge**() {

**return** collection.stream()

.mapToInt(Dragon::getAge) // Допущение, что у класса Dragon есть метод getAge

.average() // Вычисляем среднее значение, результат OptionalDouble

.orElse(Double.NaN); // Возвращаем NaN, если коллекция пуста

}

/\*\*

\* Группирует элементы коллекции по дате создания.

\* @return Строковое представление группировки.

\*/

**public** Map<LocalDateTime, List<Dragon>> **groupCountingByCreationDate**() {

**return** collection.stream()

.collect(Collectors.groupingBy(Dragon::getCreationDate));

}

/\*\*

\* Возвращает уникальные цвета драконов в коллекции.

\* @return набор уникальных цветов драконов.

\*/

**public** Set<Color> **getUniqueColors**() {

**return** collection.stream()

.map(Dragon::getColor)

.collect(Collectors.toSet());

}

/\*\*

\* Инициализирует коллекцию, загружая данные из dumpManager.

\* @return true, если инициализация прошла успешно.

\*/

**public** **boolean** **init**() {

collection.clear();

collection = dumpManager.readCollection();

lastInitTime = LocalDateTime.now();

**for** (Dragon dragon : collection) {

**int** id = dragon.getId();

**if** (id > currentId) {

currentId = id;

}

}

update();

**return** **true**;

}

/\*\*

\* Проверяет валидность всех элементов коллекции. В случае обнаружения невалидного элемента выводит ошибку.

\* @param console консоль для вывода сообщений.

\*/

**public** **void** **validateAll**(Console console) {

collection.forEach(dragon -> {

**if** (!dragon.validate()) {

console.printError("Дракон с id=" + dragon.getId() + " имеет невалидные поля.");

}

});

console.println("Выполнена проверка корректности загруженных данных");

}

/\*\*

\* Сохраняет текущую коллекцию с помощью dumpManager.

\*/

**public** **void** **saveCollection**() {

dumpManager.writeCollection(collection);

lastSaveTime = LocalDateTime.now();

}

/\*\*

\* Возвращает строковое представление коллекции.

\* @return строковое представление коллекции.

\*/

**@Override**

**public** String **toString**() {

**if** (collection.isEmpty()) **return** "Коллекция пуста!";

StringBuilder info = **new** StringBuilder();

**for** (Dragon dragon : collection) {

info.append(dragon).append("\n\n");

}

**return** info.toString().trim();

}

}

# main/java/org/lab5/managers/CommandManager.java

**package** **org.lab5.managers**;

**import** **org.lab5.commands.Command**;

**import** **java.util.ArrayList**;

**import** **java.util.HashMap**;

**import** **java.util.List**;

**import** **java.util.Map**;

/\*\*

\* Менеджер команд.

\*/

**public** **class** **CommandManager** {

**private** **final** Map<String, Command> commands = **new** HashMap<>();

**private** **final** List<String> commandHistory = **new** ArrayList<>();

/\*\*

\* Добавляет команду.

\* @param commandName Название команды.

\* @param command Команда.

\*/

**public** **void** **register**(String commandName, Command command) {

commands.put(commandName, command);

}

/\*\*

\* @return Словарь команд.

\*/

**public** Map<String, Command> **getCommands**() {

**return** commands;

}

/\*\*

\* @return История команд.

\*/

**public** List<String> **getCommandHistory**() {

**return** commandHistory;

}

/\*\*

\* Добавляет команду в историю.

\* @param command Команда.

\*/

**public** **void** **addToHistory**(String command) {

commandHistory.add(command);

}

}

# main/java/org/lab5/managers/DumpManager.java

**package** **org.lab5.managers**;

**import** **com.google.gson.\***;

**import** **com.google.gson.reflect.TypeToken**;

**import** **org.lab5.models.Dragon**;

**import** **org.lab5.utils.adapters.LocalDateTimeAdapter**;

**import** **org.lab5.utils.adapters.ZonedDateTimeTypeAdapter**;

**import** **org.lab5.utils.console.Console**;

**import** **java.io.\***;

**import** **java.nio.charset.StandardCharsets**;

**import** **java.time.LocalDateTime**;

**import** **java.time.ZonedDateTime**;

**import** **java.time.format.DateTimeFormatter**;

**import** **java.util.Collection**;

**import** **java.util.NoSuchElementException**;

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

**import** **java.util.Vector**;

/\*\*

\* Менеджер, отвечающий за работу с файлами

\*/

**public** **class** **DumpManager** {

**private** **final** Gson gson = **new** GsonBuilder()

.setPrettyPrinting()

.serializeNulls()

.registerTypeAdapter(LocalDateTime.class, **new** LocalDateTimeAdapter())

.registerTypeAdapter(ZonedDateTime.class, **new** ZonedDateTimeTypeAdapter())

.create();

**private** **final** String fileName;

**private** **final** Console console;

/\*\*

\* @param fileName

\* @param console

\*/

**public** **DumpManager**(String fileName, Console console) {

**this**.fileName = fileName;

**this**.console = console;

}

/\*\*

\* Записывает коллекцию в файл.

\* @param collection коллекция

\*/

**public** **void** **writeCollection**(Collection<Dragon> collection) {

**try** (OutputStreamWriter writer = **new** OutputStreamWriter(**new** FileOutputStream(**new** File(fileName)), StandardCharsets.UTF\_8)) {

writer.write(gson.toJson(collection));

console.println("Коллекция успешна сохранена в файл!");

} **catch** (IOException exception) {

console.printError("Загрузочный файл не может быть открыт!");

}

}

/\*\*

\* Считывает коллекцию из файл.

\* @return Считанная коллекция

\*/

**public** Vector<Dragon> **readCollection**() {

**if** (fileName != **null** && !fileName.isEmpty()) {

**try** (**var** scanner = **new** Scanner(**new** File(fileName))) {

**var** jsonString = **new** StringBuilder();

**while**(scanner.hasNextLine()) {

String line = scanner.nextLine().trim();

**if** (!line.isEmpty()) {

jsonString.append(line);

}

}

**if** (jsonString.length() == **0**) {

jsonString = **new** StringBuilder("[]");

}

Vector<Dragon> collection;

collection = gson.fromJson(jsonString.toString(), **new** TypeToken<Vector<Dragon>>() {}.getType());

System.out.println(gson.toJson(collection));

console.println("Коллекция успешно загружена!");

**return** collection;

} **catch** (FileNotFoundException exception) {

console.printError("Загрузочный файл не найден!");

} **catch** (NoSuchElementException exception) {

console.printError("Загрузочный файл пуст!");

} **catch** (JsonParseException exception) {

console.printError("В загрузочном файле не обнаружена необходимая коллекция!");

} **catch** (IllegalStateException exception) {

console.printError("Непредвиденная ошибка!");

System.exit(**0**);

}

} **else** {

console.printError("Аргумент командной строки с загрузочным файлом не найден!");

}

**return** **new** Vector<>();

}

}

# main/java/org/lab5/models/Color.java

**package** **org.lab5.models**;

/\*\*

\* Перечисление доступных цветов.

\*/

**public** **enum** Color {

RED,

YELLOW,

ORANGE;

}

# main/java/org/lab5/models/Coordinates.java

**package** **org.lab5.models**;

/\*\*

\* Класс, представляющий координаты в двумерном пространстве.

\*/

**public** **class** **Coordinates** {

**private** **double** x;

**private** Long y; // Поле не может быть null

/\*\*

\* Конструктор класса Coordinates, создает объект координат с указанными значениями.

\* @param x координата по оси X.

\* @param y координата по оси Y. Не может быть null.

\*/

**public** **Coordinates**(**double** x, Long y) {

**this**.x = x;

**this**.y = y;

}

/\*\*

\* Валидация координат. Проверяет, что значение координаты Y не null.

\* @return true, если координаты валидны, иначе - false.

\*/

**public** **boolean** **validate**() {

**if** (y == **null**) **return** **false**; // y не должен быть null

**return** **true**;

}

/\*\*

\* Возвращает строковое представление объекта в формате "x;y".

\* @return строковое представление координат.

\*/

**@Override**

**public** String **toString**() {

**return** x + ";" + y;

}

}

# main/java/org/lab5/models/Dragon.java

**package** **org.lab5.models**;

**import** **org.lab5.utils.Element**;

**import** **java.time.LocalDateTime**;

**import** **java.util.Objects**;

**import** **java.util.concurrent.atomic.AtomicInteger**;

/\*\*

\* Класс, представляющий дракона.

\*/

**public** **class** **Dragon** **extends** Element **implements** Comparable<Dragon> {

**private** **int** id;

**private** String name;

**private** Coordinates coordinates;

**private** **final** LocalDateTime creationDate; // Генерация даты создания

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

**private** Boolean speaking;

**private** Color color;

**private** DragonCharacter character;

**private** Person killer;

/\*\*

\* Конструктор класса Dragon.

\* @param id идентификатор дракона.

\* @param name имя дракона.

\* @param coordinates координаты дракона.

\* @param age возраст дракона.

\* @param speaking говорит ли дракон.

\* @param color цвет дракона.

\* @param character характер дракона.

\* @param killer убийца дракона.

\*/

**public** **Dragon**(**int** id, String name, Coordinates coordinates, **int** age, Boolean speaking, Color color, DragonCharacter character, Person killer) {

**this**.id = id;

**this**.name = name;

**this**.coordinates = coordinates;

**this**.creationDate = LocalDateTime.now(); // Автоматическая генерация даты создания

**this**.age = age;

**this**.speaking = speaking;

**this**.color = color;

**this**.character = character;

**this**.killer = killer;

}

/\*\*

\* Возвращает ID дракона.

\* @return ID дракона.

\*/

**@Override**

**public** **int** **getId**() {

**return** id;

}

/\*\*

\* Устанавливает ID дракона.

\* @param ID новый идентификатор дракона.

\*/

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

**this**.id = id;

}

/\*\*

\* Проверяет валидность данных дракона.

\* @return true, если данные валидны, иначе - false.

\*/

**public** **boolean** **validate**() {

**if** (id <= **0**) **return** **false**;

**if** (name == **null** || name.isEmpty()) **return** **false**;

**if** (coordinates == **null** || !coordinates.validate()) **return** **false**;

**if** (creationDate == **null**) **return** **false**;

**if** (age <= **0**) **return** **false**;

**if** (color == **null**) **return** **false**;

**if** (killer != **null** && !killer.validate()) **return** **false**;

// Остальные поля не требуют валидации в данном контексте

**return** **true**;

}

**public** String **getName**() {

**return** name;

}

**public** **void** **setName**(String name) {

**this**.name = name;

}

**public** Coordinates **getCoordinates**() {

**return** coordinates;

}

**public** **void** **setCoordinates**(Coordinates coordinates) {

**this**.coordinates = coordinates;

}

**public** LocalDateTime **getCreationDate**() {

**return** creationDate;

}

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

**return** age;

}

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

**this**.age = age;

}

**public** Boolean

**getSpeaking**() {

**return** speaking;

}

**public** **void** **setSpeaking**(Boolean speaking) {

**this**.speaking = speaking;

}

**public** Color **getColor**() {

**return** color;

}

**public** **void** **setColor**(Color color) {

**this**.color = color;

}

**public** DragonCharacter **getCharacter**() {

**return** character;

}

**public** **void** **setCharacter**(DragonCharacter character) {

**this**.character = character;

}

**public** Person **getKiller**() {

**return** killer;

}

**public** **void** **setKiller**(Person killer) {

**this**.killer = killer;

}

/\*\*

\* Сравнивает текущего дракона с другим.

\* @param other другой дракон.

\* @return результат сравнения.

\*/

**@Override**

**public** **int** **compareTo**(Dragon other) {

**return** Integer.compare(**this**.id, other.id);

}

/\*\*

\* Проверяет равенство текущего дракона с другим объектом.

\* @param o другой объект.

\* @return true, если объекты равны, иначе - false.

\*/

**@Override**

**public** **boolean** **equals**(Object o) {

**if** (**this** == o) **return** **true**;

**if** (o == **null** || getClass() != o.getClass()) **return** **false**;

Dragon that = (Dragon) o;

**return** Objects.equals(id, that.id);

}

/\*\*

\* Возвращает хеш-код дракона.

\* @return хеш-код.

\*/

**@Override**

**public** **int** **hashCode**() {

**return** Objects.hash(id, name, creationDate, color);

}

}

# main/java/org/lab5/models/DragonCharacter.java

**package** **org.lab5.models**;

/\*\*

\* Перечисление возможных характеров дракона.

\*/

**public** **enum** DragonCharacter {

CUNNING,

WISE,

CHAOTIC\_EVIL;

}

# main/java/org/lab5/models/Location.java

**package** **org.lab5.models**;

/\*\*

\* Класс, представляющий местоположение в трехмерном пространстве.

\*/

**public** **class** **Location** {

**private** Float x; // Поле не может быть null

**private** **float** y;

**private** Long z; // Поле не может быть null

/\*\*

\* Конструктор класса Location, создает объект местоположения с указанными координатами.

\* @param x координата по оси X. Не может быть null.

\* @param y координата по оси Y.

\* @param z координата по оси Z. Не может быть null.

\*/

**public** **Location**(Float x, **float** y, Long z) {

**this**.x = x;

**this**.y = y;

**this**.z = z;

}

/\*\*

\* Валидация местоположения. Проверяет, что значения координат X и Z не null.

\* @return true, если местоположение валидно, иначе - false.

\*/

**public** **boolean** **validate**() {

**if** (x == **null**) **return** **false**; // x не должно быть null

**if** (z == **null**) **return** **false**; // z не должно быть null

**return** **true**;

}

}

# main/java/org/lab5/models/Person.java

**package** **org.lab5.models**;

/\*\*

\* Класс, представляющий человека.

\*/

**public** **class** **Person** {

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

**private** java.time.ZonedDateTime birthday; // Поле может быть null

**private** Float height; // Поле может быть null, значение должно быть больше 0

**private** Long weight; // Поле может быть null, значение должно быть больше 0

**private** Location location; // Поле может быть null

/\*\*

\* Конструктор класса Person, создает объект человека с указанными значениями.

\* @param name имя человека. Не может быть null или пустым.

\* @param birthday день рождения человека. Может быть null.

\* @param height рост человека. Может быть null, если задан, должен быть больше 0.

\* @param weight вес человека. Может быть null, если задан, должен быть больше 0.

\* @param location местоположение человека. Может быть null.

\*/

**public** **Person**(String name, java.time.ZonedDateTime birthday, Float height, Long weight, Location location) {

**this**.name = name;

**this**.birthday = birthday;

**this**.height = height;

**this**.weight = weight;

**this**.location = location;

}

/\*\*

\* Валидация данных человека. Проверяет, что имя не null и не пустое, а также что рост и вес, если они заданы, больше 0.

\* @return true, если данные валидны, иначе - false.

\*/

**public** **boolean** **validate**() {

**if** (name == **null** || name.isEmpty()) **return** **false**; // name не должно быть null или пустым

**if** (height != **null** && height <= **0**) **return** **false**; // height, если задано, должно быть больше 0

**if** (weight != **null** && weight <= **0**) **return** **false**; // weight, если задано, также должно быть больше 0

// Проверка на location не требуется, так как оно может быть null

**return** **true**;

}

}

# main/java/org/lab5/utils/Ask.java

**package** **org.lab5.utils**;

**import** **org.lab5.models.\***;

**import** **org.lab5.utils.console.Console**;

**import** **java.util.NoSuchElementException**;

/\*\*

\* Класс для запроса данных от пользователя.

\*/

**public** **class** **Ask** {

// Исключение, выбрасываемое для выхода из процедуры ввода

**public** **static** **class** **AskBreak** **extends** Exception {}

/\*\*

\* Запрашивает данные и создает экземпляр Dragon.

\* @param console консоль для ввода данных.

\* @param id идентификатор дракона.

\* @return объект Dragon с запрошенными данными.

\* @throws AskBreak если процесс ввода был прерван.

\*/

**public** **static** Dragon **askDragon**(Console console, **int** id) **throws** AskBreak {

**try** {

// Запрос имени дракона

String name = askString(console, "Enter dragon's name: ", **true**);

// Запрос координат дракона

Coordinates coordinates = askCoordinates(console);

// Запрос возраста дракона

**int** age = askInt(console, "Enter dragon's age (>0): ", **true**);

// Запрос говорящего дракона или нет

Boolean speaking = askBoolean(console, "Is the dragon speaking? (true/false): ");

// Запрос цвета дракона

Color color = askColor(console);

// Запрос характера дракона

DragonCharacter character = askCharacter(console, "Enter dragon's character or leave blank if unknown (CUNNING/WISE/CHAOTIC\_EVIL): ");

// Запрос убийцы дракона (предполагаем, что для класса Person также реализован метод askPerson)

Person killer = askPerson(console);

**return** **new** Dragon(id, name, coordinates, age, speaking, color, character, killer);

} **catch** (NoSuchElementException | IllegalStateException e) {

console.printError("An error occurred while reading input.");

**return** **null**;

}

}

/\*\*

\* Запрашивает строку от пользователя.

\* @param console консоль для ввода.

\* @param prompt приглашение к вводу.

\* @param nonEmpty должна ли строка быть непустой.

\* @return введенная строка.

\* @throws AskBreak если процесс ввода был прерван.

\*/

**private** **static** String **askString**(Console console, String prompt, **boolean** nonEmpty) **throws** AskBreak {

String input;

**while** (**true**) {

console.print(prompt);

input = console.readln().trim();

**if** (input.equalsIgnoreCase("exit")) **throw** **new** AskBreak();

**if** (nonEmpty && input.isEmpty()) {

console.printError("The input cannot be empty. Please try again.");

} **else** {

**break**;

}

}

**return** input;

}

/\*\*

\* Запрашивает координаты от пользователя и создает объект Coordinates.

\* @param console консоль для ввода.

\* @return объект Coordinates с запрошенными координатами.

\* @throws AskBreak если процесс ввода был прерван.

\*/

**private** **static** Coordinates **askCoordinates**(Console console) **throws** AskBreak {

**double** x;

Long y;

**while** (**true**) {

x = askDouble(console, "Enter coordinate x: ", **false**);

y = askLong(console, "Enter coordinate y (non-null): ", **true**);

**if** (y != **null**) **break**;

}

**return** **new** Coordinates(x, y);

}

/\*\*

\* Запрашивает целое число от пользователя.

\* @param console консоль для ввода.

\* @param prompt приглашение к вводу.

\* @param positiveOnly должно ли число быть положительным.

\* @return введенное целое число.

\* @throws AskBreak если процесс ввода был прерван.

\*/

**private** **static** **int** **askInt**(Console console, String prompt, **boolean** positiveOnly) **throws** AskBreak {

**int** value;

**while** (**true**) {

**try** {

value = Integer.parseInt(askString(console, prompt, **true**));

**if** (positiveOnly && value <= **0**) {

console.printError("The value must be greater than 0. Please try again.");

} **else** {

**break**;

}

} **catch** (NumberFormatException e) {

console.printError("Invalid integer format. Please try again.");

}

}

**return** value;

}

/\*\*

\* Запрашивает булевое значение от пользователя.

\* @param console консоль для ввода.

\* @param prompt приглашение к вводу.

\* @return введенное булевое значение.

\* @throws AskBreak если процесс ввода был прерван.

\*/

**private** **static** **double** **askDouble**(Console console, String prompt, **boolean** positiveOnly) **throws** AskBreak {

**double** value;

**while** (**true**) {

**try** {

value = Double.parseDouble(askString(console, prompt, **true**));

**if** (positiveOnly && value <= **0**) {

console.printError("The value must be greater than 0. Please try again.");

} **else** {

**break**;

}

} **catch** (NumberFormatException e) {

console.printError("Invalid double format. Please try again.");

}

}

**return** value;

}

/\*\*

\* Запрашивает длинное целое число от пользователя.

\* @param console консоль для ввода.

\* @param prompt приглашение к вводу.

\* @param nonNullOnly должно ли число быть не null.

\* @return введенное длинное целое число.

\* @throws AskBreak если процесс ввода был прерван.

\*/

**private** **static** Long **askLong**(Console console, String prompt, **boolean** nonNullOnly) **throws** AskBreak {

Long value;

**while** (**true**) {

**try** {

value = Long.valueOf(askString(console, prompt, nonNullOnly));

**break**;

} **catch** (NumberFormatException e) {

**if** (nonNullOnly) {

console.printError("Invalid float format. Please try again.");

} **else** {

**return** **null**;

}

}

}

**return** value;

}

/\*\*

\* Запрашивает булевое значение от пользователя.

\* @param console консоль для ввода.

\* @param prompt приглашение к вводу.

\* @return введенное булевое значение.

\* @throws AskBreak если процесс ввода был прерван.

\*/

**private** **static** Boolean **askBoolean**(Console console, String prompt) **throws** AskBreak {

Boolean value;

**while** (**true**) {

String input = askString(console, prompt, **false**);

**if** (input.equalsIgnoreCase("true") || input.equalsIgnoreCase("yes")) {

value = **true**;

**break**;

} **else** **if** (input.equalsIgnoreCase("false") || input.equalsIgnoreCase("no")) {

value = **false**;

**break**;

} **else** **if** (input.isEmpty()) {

value = **null**;

**break**;

} **else** {

console.printError("Invalid boolean format. Please enter true/false or yes/no.");

}

}

**return** value;

}

/\*\*

\* Запрашивает цвет от пользователя и создает объект Color.

\* @param console консоль для ввода.

\* @return объект Color с запрошенным цветом.

\* @throws AskBreak если процесс ввода был прерван.

\*/

**private** **static** Color **askColor**(Console console) **throws** AskBreak {

Color color;

**while** (**true**) {

String input = askString(console, "Enter dragon's color (RED/YELLOW/ORANGE): ", **true**);

**try** {

color = Color.valueOf(input.toUpperCase());

**break**;

} **catch** (IllegalArgumentException e) {

console.printError("Invalid color. Please choose from RED, YELLOW, or ORANGE.");

}

}

**return** color;

}

/\*\*

\* Запрашивает характер дракона от пользователя и создает объект DragonCharacter.

\* @param console консоль для ввода.

\* @param prompt приглашение к вводу.

\* @return объект DragonCharacter с запрошенным характером.

\* @throws AskBreak если процесс ввода был прерван.

\*/

**private** **static** DragonCharacter **askCharacter**(Console console, String prompt) **throws** AskBreak {

DragonCharacter character = **null**;

String input;

**while** (**true**) {

console.println(prompt);

input = console.readln().trim();

**if** (input.equalsIgnoreCase("exit")) **throw** **new** AskBreak();

**if** (!input.isEmpty()) {

**try** {

character = DragonCharacter.valueOf(input.toUpperCase());

**break**;

} **catch** (IllegalArgumentException e) {

console.printError("Invalid character. Please choose from CUNNING/WISE/CHAOTIC\_EVIL");

}

} **else** {

**break**;

}

}

**return** character;

}

/\*\*

\* Запрашивает информацию о человеке от пользователя и создает объект Person.

\* @param console консоль для ввода.

\* @return объект Person с запрошенной информацией.

\* @throws AskBreak если процесс ввода был прерван.

\*/

**private** **static** Person **askPerson**(Console console) **throws** AskBreak {

String name;

java.time.ZonedDateTime birthday;

Float height;

Long weight;

Location location;

console.println("Enter dragon's killer information or leave blank if unknown:");

name = askString(console, "Killer's name: ", **false**);

**if** (name.isEmpty()) **return** **null**; // предполагаем, что убийца может быть не указан

birthday = askDateTime(console, "Killer's birthday (enter in format: YYYY-MM-DDThh:mm:ss+HH:MM) or leave blank: ");

height = askFloat(console, "Killer's height (>0) or leave blank: ", **true**);

weight = askLong(console, "Killer's weight (>0) or leave blank: ", **true**);

location = askLocation(console); // предполагаем, что реализован метод askLocation

**return** **new** Person(name, birthday, height, weight, location);

}

/\*\*

\* Запрашивает дату и время от пользователя и создает объект ZonedDateTime.

\* @param console консоль для ввода.

\* @param prompt приглашение к вводу.

\* @return объект ZonedDateTime с запрошенной датой и временем.

\* @throws AskBreak если процесс ввода был прерван.

\*/

**private** **static** java.time.ZonedDateTime **askDateTime**(Console console, String prompt) **throws** AskBreak {

java.time.ZonedDateTime dateTime = **null**;

**while** (**true**) {

String input = askString(console, prompt, **false**);

**if** (input.isEmpty()) {

**return** **null**;

}

**try** {

dateTime = java.time.ZonedDateTime.parse(input); // Предполагаем стандартный формат ISO 8601

**break**;

} **catch** (java.time.format.DateTimeParseException e) {

// 2024-02-07T16:15:24+00:00

console.printError("Invalid date-time format. Please use format: YYYY-MM-DDThh:mm:ss+HH:MM");

}

}

**return** dateTime;

}

/\*\*

\* Запрашивает число с плавающей точкой от пользователя.

\* @param console консоль для ввода.

\* @param prompt приглашение к вводу.

\* @param positiveOnly должно ли число быть положительным.

\* @return введенное число с плавающей точкой.

\* @throws AskBreak если процесс ввода был прерван.

\*/

**private** **static** Float **askFloat**(Console console, String prompt, **boolean** positiveOnly) **throws** AskBreak {

Float value = **null**;

**while** (**true**) {

String input = askString(console, prompt, **false**);

**if** (input.isEmpty()) {

**return** **null**;

}

**try** {

value = Float.parseFloat(input);

**if** (positiveOnly && value <= **0**) {

console.printError("The value must be greater than 0. Please try again.");

} **else** {

**break**;

}

} **catch** (NumberFormatException e) {

console.printError("Invalid float format. Please try again.");

}

}

**return** value;

}

/\*\*

\* Запрашивает у пользователя данные о местоположении и создает объект Location.

\* @param console консоль для ввода данных.

\* @return объект Location с запрошенными данными.

\* @throws AskBreak если процесс ввода был прерван.

\*/

**private** **static** Location **askLocation**(Console console) **throws** AskBreak {

console.println("Enter location details:");

Float x = askFloat(console, "Location.x (non-null): ", **false**);

**float** y = (**float**) askDouble(console, "Location.y: ", **false**); // y - примитивный тип, null не допускается

Long z = askLong(console, "Location.z (non-null): ", **true**);

**if** (x == **null** || z == **null**) {

console.printError("Location coordinates cannot be null. Please enter valid values.");

**throw** **new** AskBreak();

}

**return** **new** Location(x, y, z);

}

}

# main/java/org/lab5/utils/Element.java

**package** **org.lab5.utils**;

/\*\*

\* Абстрактный класс Element, представляющий элемент с идентификатором.

\*/

**public** **abstract** **class** **Element**{

/\*\*

\* Возвращает идентификатор элемента.

\* @return идентификатор элемента.

\*/

**public** **abstract** **int** **getId**();

}

# main/java/org/lab5/utils/ExecutionResponse.java

**package** **org.lab5.utils**;

/\*\*

\* Класс, представляющий ответ на выполнение команды.

\*/

**public** **class** **ExecutionResponse** {

**private** **boolean** exitCode;

**private** String message;

/\*\*

\* Конструктор класса ExecutionResponse.

\* @param code код завершения команды.

\* @param s сообщение об результате выполнения команды.

\*/

**public** **ExecutionResponse**(**boolean** code, String s) {

exitCode = code;

message = s;

}

/\*\*

\* Конструктор класса ExecutionResponse, создает успешный ответ с указанным сообщением.

\* @param s сообщение об успешном выполнении команды.

\*/

**public** **ExecutionResponse**(String s) {

**this**(**true**, s);

}

/\*\*

\* Возвращает код завершения команды.

\* @return код завершения команды.

\*/

**public** **boolean** **getExitCode**() {

**return** exitCode;

}

/\*\*

\* Возвращает сообщение об результате выполнения команды.

\* @return сообщение об результате выполнения команды.

\*/

**public** String **getMessage**() {

**return** message;

}

/\*\*

\* Возвращает строковое представление объекта.

\* @return строковое представление объекта в формате "exitCode;message;".

\*/

**public** String **toString**() { **return** String.valueOf(exitCode)+";"+message+";"; }

}

# main/java/org/lab5/utils/Runner.java

**package** **org.lab5.utils**;

**import** **java.io.File**;

**import** **java.io.FileNotFoundException**;

**import** **java.util.ArrayList**;

**import** **java.util.List**;

**import** **java.util.NoSuchElementException**;

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

**import** **org.lab5.exceptions.ScriptRecursionException**;

**import** **org.lab5.managers.CommandManager**;

**import** **org.lab5.utils.console.Console**;

/\*\*

\* Класс для выполнения команд в различных режимах.

\*/

**public** **class** **Runner** {

**public** **enum** ExitCode {

OK,

ERROR,

EXIT,

}

**private** **final** Console console;

**private** **final** CommandManager commandManager;

**private** **final** List<String> scriptStack = **new** ArrayList<>();

/\*\*

\* Конструктор класса Runner.

\* @param console

\* @param commandManager

\*/

**public** **Runner**(Console console, CommandManager commandManager) {

**this**.console = console;

**this**.commandManager = commandManager;

}

/\*\*

\* Интерактивный режим

\*/

**public** **void** **interactiveMode**() {

//var userScanner = Interrogator.getUserScanner();

**try** {

ExitCode commandStatus;

String[] userCommand = {"", ""};

**do** {

console.prompt();

userCommand = (console.readln().trim() + " ").split(" ", **2**);

userCommand[**1**] = userCommand[**1**].trim();

commandManager.addToHistory(userCommand[**0**]);

commandStatus = launchCommand(userCommand);

} **while** (commandStatus != ExitCode.EXIT);

} **catch** (NoSuchElementException exception) {

console.printError("Пользовательский ввод не обнаружен!");

} **catch** (IllegalStateException exception) {

console.printError("Непредвиденная ошибка!");

}

}

/\*\*

\* Режим для запуска скрипта.

\* @param argument Аргумент скрипта

\* @return Код завершения.

\*/

**public** ExitCode **scriptMode**(String argument) {

String[] userCommand = {"", ""};

ExitCode commandStatus;

scriptStack.add(argument);

**try** (Scanner scriptScanner = **new** Scanner(**new** File(argument))) {

**if** (!scriptScanner.hasNext()) **throw** **new** NoSuchElementException();

console.selectFileScanner(scriptScanner);

**do** {

userCommand = (scriptScanner.nextLine().trim() + " ").split(" ", **2**);

userCommand[**1**] = userCommand[**1**].trim();

**while** (scriptScanner.hasNextLine() && userCommand[**0**].isEmpty()) {

userCommand = (scriptScanner.nextLine().trim() + " ").split(" ", **2**);

userCommand[**1**] = userCommand[**1**].trim();

}

console.println(console.getPrompt() + String.join(" ", userCommand));

**if** (userCommand[**0**].equals("execute\_script")) {

**for** (String script : scriptStack) {

**if** (userCommand[**1**].equals(script))

{

console.selectConsoleScanner();

**throw** **new** ScriptRecursionException();

}

}

}

commandStatus = launchCommand(userCommand);

} **while** (commandStatus == ExitCode.OK && scriptScanner.hasNextLine());

console.selectConsoleScanner();

**if** (commandStatus == ExitCode.ERROR && !(userCommand[**0**].equals("execute\_script") && !userCommand[**1**].isEmpty())) {

console.println("Проверьте скрипт на корректность введенных данных!");

}

**return** commandStatus;

} **catch** (FileNotFoundException exception) {

console.printError("Файл со скриптом не найден!");

} **catch** (NoSuchElementException exception) {

console.printError("Файл со скриптом пуст!");

} **catch** (ScriptRecursionException exception) {

console.printError("Скрипты не могут вызываться рекурсивно!");

} **catch** (IllegalStateException exception) {

console.printError("Непредвиденная ошибка!");

System.exit(**0**);

} **finally** {

scriptStack.remove(scriptStack.size() - **1**);

}

**return** ExitCode.ERROR;

}

/\*\*

\* Запускает команду.

\* @param userCommand команда для запуска.

\* @return код завершения.

\*/

**private** ExitCode **launchCommand**(String[] userCommand) {

**if** (userCommand[**0**].isEmpty()) **return** ExitCode.OK;

**var** command = commandManager.getCommands().get(userCommand[**0**]);

**if** (command == **null**) {

console.printError("Команда '" + userCommand[**0**] + "' не найдена. Наберите 'help' для справки");

**return** ExitCode.ERROR;

}

**switch** (userCommand[**0**]) {

**case** "exit" -> {

**if** (!commandManager.getCommands().get("exit").apply(userCommand).getExitCode()) **return** ExitCode.ERROR;

**else** **return** ExitCode.EXIT;

}

**case** "execute\_script" -> {

**if** (!commandManager.getCommands().get("execute\_script").apply(userCommand).getExitCode()) **return** ExitCode.ERROR;

**else** **return** scriptMode(userCommand[**1**]);

}

**default** -> { **if** (!command.apply(userCommand).getExitCode()) **return** ExitCode.ERROR; }

};

**return** ExitCode.OK;

}

}

# main/java/org/lab5/utils/adapters/LocalDateTimeAdapter.java

**package** **org.lab5.utils.adapters**;

**import** **com.google.gson.\***;

**import** **java.lang.reflect.Type**;

**import** **java.time.LocalDateTime**;

**import** **java.time.format.DateTimeFormatter**;

/\*\*

\* Класс-адаптер для сериализации и десериализации объектов LocalDateTime для библиотеки Gson.

\*/

**public** **class** **LocalDateTimeAdapter** **implements** JsonSerializer<LocalDateTime>, JsonDeserializer<LocalDateTime> {

/\*\*

\* Сериализует объект LocalDateTime в JsonElement.

\* @param date объект LocalDateTime.

\* @param typeOfSrc тип исходного объекта.

\* @param context контекст сериализации.

\* @return JsonElement, представляющий объект date.

\*/

**@Override**

**public** JsonElement **serialize**(LocalDateTime date, Type typeOfSrc, JsonSerializationContext context) {

**return** **new** JsonPrimitive(date.format(DateTimeFormatter.ISO\_LOCAL\_DATE\_TIME)); // iso 6901

}

/\*\*

\* Десериализует JsonElement в объект LocalDateTime.

\* @param json JsonElement, представляющий объект LocalDateTime.

\* @param type тип объекта, в который нужно преобразовать json.

\* @param context контекст десериализации.

\* @return объект LocalDateTime, полученный из json.

\* @throws JsonParseException если json невозможно преобразовать в LocalDateTime.

\*/

**@Override**

**public** LocalDateTime **deserialize**(JsonElement json, Type type, JsonDeserializationContext context) **throws** JsonParseException {

**return** LocalDateTime.parse(json.getAsJsonPrimitive().getAsString());

}

}

# main/java/org/lab5/utils/adapters/ZonedDateTimeTypeAdapter.java

**package** **org.lab5.utils.adapters**;

**import** **java.lang.reflect.Type**;

**import** **java.text.ParseException**;

**import** **java.time.LocalDateTime**;

**import** **java.time.ZoneId**;

**import** **java.time.ZonedDateTime**;

**import** **java.time.format.DateTimeFormatter**;

**import** **com.google.gson.JsonDeserializationContext**;

**import** **com.google.gson.JsonDeserializer**;

**import** **com.google.gson.JsonElement**;

**import** **com.google.gson.JsonParseException**;

**import** **com.google.gson.JsonPrimitive**;

**import** **com.google.gson.JsonSerializationContext**;

**import** **com.google.gson.JsonSerializer**;

/\*\*

\* Класс-адаптер для сериализации и десериализации объектов ZonedDateTime для библиотеки Gson.

\*/

**public** **class** **ZonedDateTimeTypeAdapter** **implements** JsonDeserializer<ZonedDateTime>, JsonSerializer<ZonedDateTime> {

/\*\*

\* Десериализует JsonElement в объект ZonedDateTime.

\* @param json JsonElement, представляющий объект ZonedDateTime.

\* @param typeOfT тип объекта, в который нужно преобразовать json.

\* @param context контекст десериализации.

\* @return объект ZonedDateTime, полученный из json.

\* @throws JsonParseException если json невозможно преобразовать в ZonedDateTime.

\*/

**@Override**

**public** ZonedDateTime **deserialize**(JsonElement json, Type typeOfT,

JsonDeserializationContext context) **throws** JsonParseException {

**try** {

String jsonStr = json.getAsJsonPrimitive().getAsString();

**return** parseLocalDateTime(jsonStr);

} **catch** (ParseException e) {

**throw** **new** JsonParseException(e.getMessage(), e);

}

}

/\*\*

\* Парсит строку в объект ZonedDateTime.

\* @param dateString строка с датой и временем.

\* @return объект ZonedDateTime, полученный из строки.

\* @throws ParseException если строку не удается преобразовать в ZonedDateTime.

\*/

**private** ZonedDateTime **parseLocalDateTime**(**final** String dateString) **throws** ParseException {

**if** (dateString != **null** && dateString.trim().length() > **0**) {

**if** (dateString.length() == **19**) { // whithout zone

LocalDateTime localDateTime = LocalDateTime.parse(dateString);

**return** localDateTime.atZone(ZoneId.of("America/Sao\_Paulo"));

// return localDateTime.atZone(ZoneOffset.UTC);

} **else** {

**return** ZonedDateTime.parse(dateString, DateTimeFormatter.ISO\_ZONED\_DATE\_TIME);

}

} **else** {

**return** **null**;

}

}

/\*\*

\* Сериализует объект ZonedDateTime в JsonElement.

\* @param src объект ZonedDateTime для сериализации.

\* @param typeOfSrc тип исходного объекта.

\* @param context контекст сериализации.

\* @return JsonElement, представляющий объект src.

\*/

**@Override**

**public** JsonElement **serialize**(**final** ZonedDateTime src, **final** Type typeOfSrc,

**final** JsonSerializationContext context) {

**final** String strDateTime = DateTimeFormatter.ISO\_OFFSET\_DATE\_TIME

.withZone(ZoneId.of("America/Sao\_Paulo"))

.format(src);

**return** **new** JsonPrimitive(strDateTime);

}

}

# main/java/org/lab5/utils/console/Console.java

**package** **org.lab5.utils.console**;

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

**import** **java.util.NoSuchElementException**;

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

/\*\*

\* Интерфейс для работы с консолью.

\*/

**public** **interface** **Console** {

/\*\*

\* Выводит объект на консоль без перехода на новую строку.

\* @param obj объект для вывода.

\*/

**void** **print**(Object obj);

/\*\*

\* Выводит объект на консоль с переходом на новую строку.

\* @param obj объект для вывода.

\*/

**void** **println**(Object obj);

/\*\*

\* Читает строку из консоли.

\* @return строка, считанная с консоли.

\*/

String **readln**();

/\*\*

\* Проверяет, возможно ли считывание строки из консоли.

\* @return true, если считывание строки возможно, иначе - false.

\*/

**boolean** **isCanReadln**();

/\*\*

\* Выводит сообщение об ошибке на консоль.

\* @param obj сообщение об ошибке.

\*/

**void** **printError**(Object obj);

/\*\*

\* Выводит на консоль таблицу из двух объектов.

\* @param obj1 первый объект таблицы.

\* @param obj2 второй объект таблицы.

\*/

**void** **printTable**(Object obj1, Object obj2);

/\*\*

\* Отображает приглашение к вводу на консоли.

\*/

**void** **prompt**();

/\*\*

\* Возвращает текущее приглашение к вводу.

\* @return строка приглашения к вводу.

\*/

String **getPrompt**();

/\*\*

\* Выбирает Scanner для чтения из файла.

\* @param obj объект Scanner для чтения из файла.

\*/

**void** **selectFileScanner**(Scanner obj);

/\*\*

\* Выбирает Scanner для чтения из консоли.

\*/

**void** **selectConsoleScanner**();

}

# main/java/org/lab5/utils/console/StandardConsole.java

**package** **org.lab5.utils.console**;

**import** **java.io.FileOutputStream**;

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

**import** **java.util.NoSuchElementException**;

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

/\*\*

\* Класс, реализующий интерфейс Console для работы с консолью.

\*/

**public** **class** **StandardConsole** **implements** Console {

**private** **static** **final** String PROMPT = "$ ";

**private** **static** Scanner fileScanner = **null**;

**private** **static** Scanner defScanner = **new** Scanner(System.in);

**private** FileOutputStream fileOutputStream;

/\*\*

\* Выводит объект на консоль без перехода на новую строку.

\* @param obj объект для вывода.

\*/

**@Override**

**public** **void** **print**(Object obj) {

System.out.print(obj);

}

/\*\*

\* Выводит объект на консоль с переходом на новую строку.

\* @param obj объект для вывода.

\*/

**@Override**

**public** **void** **println**(Object obj) {

System.out.println(obj);

}

/\*\*

\* Выводит сообщение об ошибке на консоль.

\* @param obj сообщение об ошибке.

\*/

**@Override**

**public** **void** **printError**(Object obj) {

System.err.println("Error: " + obj);

}

/\*\*

\* Читает строку из консоли.

\* @return строка, считанная с консоли.

\* @throws NoSuchElementException если нет элементов для чтения.

\* @throws IllegalStateException если сканер закрыт.

\*/

**@Override**

**public** String **readln**() **throws** NoSuchElementException, IllegalStateException {

**return** (fileScanner!=**null**?fileScanner:defScanner).nextLine();

}

/\*\*

\* Проверяет, возможно ли считывание строки из консоли.

\* @return true, если считывание строки возможно, иначе - false.

\* @throws IllegalStateException если сканер закрыт.

\*/

**@Override**

**public** **boolean** **isCanReadln**() **throws** IllegalStateException {

**return** (fileScanner!=**null**?fileScanner:defScanner).hasNextLine();

}

/\*\*

\* Выводит на консоль таблицу из двух объектов.

\* @param elementLeft первый объект таблицы.

\* @param elementRight второй объект таблицы.

\*/

**@Override**

**public** **void** **printTable**(Object elementLeft, Object elementRight) {

System.out.printf(" %-35s%-1s%n", elementLeft, elementRight);

}

/\*\*

\* Отображает приглашение к вводу на консоли.

\*/

**@Override**

**public** **void** **prompt**() {

print(PROMPT);

}

/\*\*

\* Возвращает текущее приглашение к вводу.

\* @return строка приглашения к вводу.

\*/

**@Override**

**public** String **getPrompt**() {

**return** PROMPT;

}

/\*\*

\* Выбирает сканер для чтения из файла.

\* @param scanner сканер для чтения из файла.

\*/

**@Override**

**public** **void** **selectFileScanner**(Scanner scanner) {

**this**.fileScanner = scanner;

}

/\*\*

\* Выбирает сканер для чтения из консоли.

\*/

**@Override**

**public** **void** **selectConsoleScanner**() {

**this**.fileScanner = **null**;

}

}

# **Выводы**:

В результате проделанной работы, я создал консольное приложение, которое реализует управление коллекцией объектов в интерактивном режиме, научился работать с различными структурами данных в Java и файлами, а также углубил свои знания о ООП в Java, изучил интерфейсы Comparator и Compareable. изучил параметризованные типы, wildcard-параметры и применил javadoc на практике.