# Java Advanced - Exam Preparation – 1

This document defines the exercises for the "Java Advanced" course @ SoftUni.

Please submit your solutions (source code) to all below-described problems in [Judge](https://judge.softuni.org/Contests/4023/Java-Advanced-Exam-Preparation-1-MLC)

## OS-Planning

*You are hired to create a program that schedules the work of an OS and avoids tasks that could harm it.*

On the **first line,** you will be given some **tasks** as **integer values**, separated by a **comma and space** **", "**. On the **second line,** you will be given some **threads as integer values**, separated by a single space**.** On the **third line**, you will receive the **integer value** of a **task** that you need to **kill**. Your job is to **stop the work of the OS as soon as you get to this task**, otherwise, your OS will crash. The **thread** that gets **first** to this task, **kills** **it**.

The **OS** **works** in the following way:

* It takes the **first given thread value** and the **last given task** **value.**
* If the **thread value** is **greater** than or **equal** to the **task** value, **the task and thread get removed**.
* If the **thread** **value** is **less** than the **task value**, the **thread** gets **removed**, but the **task** **remains**.

After you finish the needed task, print on a single line:

**"Thread with value {thread} killed task {taskToBeKilled}"**

Then print the remaining threads (**including** the **one that killed** the **task**) starting from the first on a single line, separated by a single space.

### Input

* **On the first line,** you will receive the **tasks**, separated by **", "**.
* **On the second line,** you will the **threads**, separated by a single space.
* **On the third line**, you will receive a **single integer** – a **value of the task** to be killed.

### Output

* Print the **thread that killed the task** and the **task itself** in the format given above.
* Print the **remaining threads** starting **from the first** on a single line, separated by a single space.

### Constraints

* The needed task will always be with a **unique** value.
* You will **always have enough threads to get to the needed task.**

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| 20, 23, 54, 34, 90  150 64 20 34  54 | Thread with value 20 killed task 54  20 34 | First, the thread with a value of 150 is taken and the task with a value of 90. The thread has a bigger value, so both thread and task get removed. Next, thread 64 finishes task 34 and both get removed. Then thread 20 gets to task 54 and kills it. |
| 33, 12, 15, 40, 45, 60  30 20 53 67 84 90  40 | Thread with value 90 killed task 40  90 | Thread 30 takes task 60, but the task has greater value, so the thread gets removed. Then thread 20 takes task 60 and the same happens – the thread gets removed. Then the same happens with thread 53. After that, thread 67 takes task 60 and finishes it. Then thread 84 finishes task 45. Finally, thread 90 gets to task 40, which should be killed and the program stops. |

## Re-Volt

You will be given an integer N for the size of the square matrix and then an integer for the count of commands. On the next **n** lines, you will receive the rows of the matrix. The player starts at a random position (the player is marked with **"f"**) and **all of the empty slots** will be filled with **"-"** (**dash**). The goal is to reach the finish mark which will be marked with an **"F"**. On the field, there can also be bonuses and traps. Bonuses are marked with **"B"** and traps are marked with **"T"**.

Each turn you will be given commands for **the player’s movement.** If the player **goes** **out** of the matrix, he comes in from **the other side**. For example, if the player is on 0, 0 and the next command is left, he goes to the last spot on the first row.

If the player steps on a **bonus**, he should move another step forward in the direction he is going. If the player steps on a **trap**, he should move a step backward.

When the player reaches the **finish mark** or the **count of commands is reached,** the game ends.

### Input

* On the first line, you are given the integer N – the size of the square matrix.
* On the second you are given the count of commands.
* The next N lines hold the values for every row.
* On each of the next lines, you will get commands for movement directions.

### Output

* If the player reaches the finish mark, print:
  + **"Player won!"**
* If the player reaches the commands count and hasn’t reached the finish mark print:
  + **"Player lost!"**
* At the end print the matrix.

### Constraints

* The size of the matrix will be between **[2…20].**
* The players will always be indicated with **"f".**
* If the player steps on the finish mark **at the same time** as his last command, he **wins** the game.
* Commands will be in the format of **up**, **down**, **left** or **right**.
* There won't be a case where you bypass the finish while you are on a trap or a bonus.
* A trap will never place you into a bonus or another trap and a bonus will never place you into a trap or another bonus.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 5  5  -----  -f---  -B---  --T--  -F---  down  right  down | Player won!  -----  -----  -B---  --T--  -f--- | The first command is **down** so **f** moves down. On turn 1, the player steps on a bonus and does an additional step. On turn 2, the player steps on a trap and goes a step back. After each turn the field is:  1 2 3  ----- ----- -----  ----- ----- -----  -B--- -B--- -B---  -fT-- -fT-- --T--  -F--- -F--- -f---  In turn, 3 **f** reaches the finish 'F' and wins the game. |
| 4  3  ----  -f-B  --T-  ---F  up  up  left | Player lost!  ----  ---B  --T-  f--F | The first command is **up** so **f** moves up and the new player position is **(0,1).** The next command is **up** again, so the player goes **out** **of the matrix** and **comes back from the other side at (3,1).** Then the command is **left**, so the end position is **(3,0)** which is not the finish mark, so the **player has lost**. |

## VetClinic

### Preparation

Download the skeleton provided in Judge. **Do not** change the **packages**!

**Pay attention to name the package parking, all the classes, their fields, and methods the same way they are presented in the following document. It is also important to keep the project structure as described.**

### Problem description

Your task is to create a repository, which stores items by creating the classes described below.

First, write a Java class **Pet** with the following fields:

* **name: String**
* **age: int**
* **owner: String**

The class **constructor** should receive a **name, age,** and **owner.** You need to create the appropriate **getters and setters**. The class should override the **toString()** method in the following format:

**"{name} {age} ({owner})"**

**Next**, write a Java class **Clinic** that has **data** (a collection, which stores the Pets). All entities inside the repository have the **same fields**. Also, the **Clinic** class should have those fields:

* **capacity: int**

The class **constructor** should receive **capacity**, also it should initialize the **data** with a new instance of the collection**.** Implement the following features:

* Field **data** – **List** that holds added pets
* Method add(Pet pet) – **adds** an **entity** to the data **if** **there** **is** an **empty cell** for the pet.
* Method remove(String name) – removes the pet by **given name,** if such **exists**, and **returns boolean**.
* Method **getPet(String name, String owner)** – returns the pet with the **given name** and **owner** or **null if no such pet exists**.
* Method getOldestPet() – returns the oldest Pet.
* Getter getCount – **returns** the **number** of pets.
* **getStatistics()** – **returns** a **String** in the following **format**:
  + **"The clinic has the following patients:  
    {name} {owner}  
    {name} {owner}**

**(…)**"

### Constraints

* The **combinations** of **names** and **owners** will **always be unique**.
* The **age** of the pets will always be **positive**.

### Examples

This is an example of how the **Clinic** class is **intended to be used**.

|  |
| --- |
| Sample code usage |
| // Initialize the repository  Clinic clinic = new Clinic(20);  // Initialize entity  Pet dog = new Pet("Ellias", 5, "Tim");  // Print Pet  System.out.println(dog); // Ellias 5 (Tim)  // Add Pet  clinic.add(dog);  // Remove Pet  System.out.println(clinic.remove("Ellias")); // true  System.out.println(clinic.remove("Pufa")); // false  Pet cat = new Pet("Bella", 2, "Mia");  Pet bunny = new Pet("Zak", 4, "Jon");  clinic.add(cat);  clinic.add(bunny);  // Get Oldest Pet  Pet oldestPet = clinic.getOldestPet();  System.out.println(oldestPet); // Zak 4 (Jon)  // Get Pet  Pet pet = clinic.getPet("Bella", "Mia");  System.out.println(pet); // Bella 2 (Mia)  // Count  System.out.println(clinic.getCount()); // 2  // Get Statistics  System.out.println(clinic.getStatistics());  //The clinic has the following patients:  //Bella Mia  //Zak Jon |

## Submission

Zip all the files in the project folder except the **bin** and **obj** folders

Submit a **single .zip file**, containing **vetClinic package, with the classes inside (Pet, Clinic, and the Main class)**, there is no specific content required inside the Main class e. g. you can do any kind of local testing of your program there. However, there should be a **main(String[] args)** method inside.