# Loot box

*You are tired of being the only one on your team without cool items, so you decided to buy some loot boxes which have a chance to drop some cool items.*

Every purchase gives you two loot boxes and they are represented as a sequence of integers. First, you will be given **a sequence of integers, representing the first loot box**. Afterward, you will be given another **sequence of integers representing the second loot box**.

You need to start from the **first item** in the first box and **sum** it up with the last item in the second box. If the **sum** of their values is **an even number,** add the **summed** item to **your collection of claimed items** and **remove** them **both** from the boxes. Otherwise, remove the last item from the second box and add it to the last position in the first box. You need to **stop** summing items when one of the boxes becomes empty.

If the first loot box becomes empty print:

**"First lootbox is empty"**

If the second loot box becomes empty print:

**"Second lootbox is empty"**

In the end, you need to determine the quality of your claimed items. If the sum of the claimed items is equal to or greater than 100, print:

**"Your loot was epic! Value: {sum of claimed items}"**

Else print:

**"Your loot was poor... Value: {sum of claimed items}"**

### Input

* On the **first line**, you will receive the integers representing the **first loot box**, **separated** by a **single space**.
* On the **second line**, you will receive the integers representing the **second loot box, separated** by a **single space**.

### Output

* On the **first** line of output – print which box got empty in the format described above.
* On the **second** line – the quality of your loot in the format described above.

### Constraints

* All of the given numbers will be valid integers in the range **[0, 100]**.
* There won’t be a case where both loot boxes become empty at the same time.

### Examples

|  |  |  |
| --- | --- | --- |
| ****Input**** | ****Output**** | ****Comment**** |
| **10 11 8 13 5 6**  **0 4 7 3 6 23 3** | **Second lootbox is empty**  **Your loot was poor... Value: 42** | First, we sum 10 and 3. We get 13, which is not an even number, so we take the last item from the second box and move it to the last position in the first box. The current state of the boxes:  10 11 8 13 5 6 3  0 4 7 3 6 23  The next sum is 33 so we do the same again. On the third iteration, the sum is 16 which is an even number, so we remove both of the boxes and we add the value to our claimed items. We keep summing items until one of the boxes becomes empty. |
| **20 40 60 80 100**  **10 20 30 40 50 60** | **First lootbox is empty**  **Your loot was epic! Value: 500** |  |

# 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**. |

# Guild

## Preparation

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

**Pay attention to the name of the package guild, 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 that stores players by creating the classes described below.

**Player**

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

* **name: String**
* **clazz: String**
* **rank: String – "Trial" by default**
* **description: String – "n/a" by default**

The class **constructor** should receive the **name and clazz**. You need to create the appropriate **getters and setters**. Override the **toString()** method in the following format:

**"Player {name}: {clazz}**

**Rank: {rank}**

**Description: {description}"**

**Guild**

**Next**, write a Java class **Guild** that has **a roster** (a collection that stores **Player** entities). All entities inside the repository have the **same fields**. Also, the **Guild** class should have those **fields**:

* **name: String**
* **capacity: int**

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

* Method addPlayer(Player player) - **adds** an **entity** to the roster **if** **there** **is** **room** for it
* Method removePlayer(String name) - removes a player by **given name,** if such **exists**, and **returns boolean**
* Method promotePlayer(String name) - **promote** (**set his rank to "Member"**) the **first player** with the **given name.** If the player is **already** a "Member", **do nothing.**
* Method **demotePlayer(String name)- demote (set his rank to "Trial")** the first player with the **given** name. If the player is **already** a "Trial",  **do nothing.**
* Method kickPlayersByClass(String clazz) - removes all the players by the given class and returns **all removed players** from that **class as an array**
* Method count() - **returns** the **number** of players
* Method **report()** - **returns** a **String** in the following **format:**
  + **"Players in the guild: {guildName}:  
    {Player1}  
    {Player2}  
    (…)"**

## Constraints

* The **names** of the players will be **always unique**.
* You will always have a player added before receiving methods manipulating the Guild's players.

## Examples

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

|  |
| --- |
| Sample code usage |
| package guild;  public class Main {  public static void main(String[] args) {  //Initialize the repository (guild)  Guild guild = new Guild("Weekend Raiders", 20);  //Initialize entity  Player player = new Player("Mark", "Rogue");  //Print player  System.*out*.println(player);  //Player Mark: Rogue  //Rank: Trial  //Description: n/a   //Add player  guild.addPlayer(player);  System.*out*.println(guild.count()); //1  System.*out*.println(guild.removePlayer("Gosho")); //false   Player firstPlayer = new Player("Pep", "Warrior");  Player secondPlayer = new Player("Lizzy", "Priest");  Player thirdPlayer = new Player("Mike", "Rogue");  Player fourthPlayer = new Player("Marlin", "Mage");   //Add description to player  secondPlayer.setDescription("Best healer EU");   //Add players  guild.addPlayer(firstPlayer);  guild.addPlayer(secondPlayer);  guild.addPlayer(thirdPlayer);  guild.addPlayer(fourthPlayer);   //Promote player  guild.promotePlayer("Lizzy");   //Remove Player  System.*out*.println(guild.removePlayer("Pep")); //true   Player[] kickedPlayers = guild.kickPlayersByClass("Rogue");  for (Player kickedPlayer : kickedPlayers) {  System.*out*.print(kickedPlayer.getName() + " ");  }  //Mark Mike   System.*out*.println(guild.report());  //Players in the guild: Weekend Raiders:  //Player Lizzy: Priest  //Rank: Member  //Description: Best healer EU  //Player Marlin: Mage  //Rank: Trial  //Description: n/a  } } |