**MRA - Mars Rover API**

# **Goal**

MRA (Mars Rover API) is an API to move a rover (i.e., a planet exploration vehicle) and keeps track of its position and direction, together with the obstacles that it has encountered (if any) while taking a tour on a planet.

The planet, where the rover moves, is represented as a grid with *x* and *y* coordinates. The planet may contain obstacles in its cells.

The rover starts its journey from the landing position—ie., at coordinates *(0,0)*—, facing North.

To let the rover move on the planet, the rover receives a command string—it can contain a single command or a combination of single commands. Once the rover has executed the commands (specified in the command string), it returns a string containing its new position and direction, together with the obstacles it has encountered (if any).

So far, you do not need to know more. Further details will be given in the User Stories section.

# **Instructions**

**READ** the instructions carefully.

**FORK** the repository of MRA and make sure your forked repository is **PUBLIC**. Then, **CLONE** the repository and **IMPORT** the project into Eclipse.

You are asked to **DEVELOP** and **TEST** MRA in an incremental way; namely, by tackling a user story at time. In the user story, you are asked to implement (and review the implementation of) the following method (see the Eclipse project):

* **MarsRover.executeCommand(String commandString)**.

You **CANNOT:**

* change the signature (i.e., method name, parameter types, and return type) of the provided methods;
* move the provided methods to other classes;
* change the name of the provided classes.

You **CAN** add fields, methods (e.g., methods used by tests to set up the fixture or methods used by the provided methods), or even classes (including other test classes), as long as you comply with the provided API (see the API Usage section and the Eclipse project).

You **DO NOT** need to develop a GUI.

The MRA requirements are divided into a set of **USER STORIES**, which serve as a to-do list. You should be able to incrementally develop MRA without an upfront comprehension of all the MRA requirements. **DO NOT** read ahead, and handle the requirements (i.e., specified in the user stories) one at a time in the order provided. Develop MRA by starting from the first story’s requirement. When a story is **IMPLEMENTED**, move on to the **NEXT** one. A story is implemented when you are confident that your program correctly implements all the functionality stipulated by the story’s requirement.

How can I be confident that my program correctly implements all the functionality stipulated by the story’s requirement? **DESIGN** **TEST CASES** for that user story by using the techniques you know and then run the test cases.

Also, run your **REGRESSION TEST SUITE** after your changes to ensure the program does not regress.

Note that you may need to review your program as you progress towards more advanced requirements.

Tip: commit (at least) each time a story is implemented

If you need to handle error situations (including situations unspecified by the user stories), throw a **MarsRoverException**.

At the end of the task, you are asked to provide the link to your repository. Ensure that your repository is **PUBLIC**.

# **API Usage**

Take some minutes to understand, in broad terms, how the API works (see also the JavaDoc comments of the provided methods in the Eclipse project). If you do not fully understand the API, do not worry because further details will be given later in the User Stories section.

A typical API usage follows.

**// It initializes the rover at the coordinates (0,0), facing North, on a**

**// 10x12 planet with obstacles at the coordinates (5,5) and (7,8).**

**List<String> planetObstacles = new ArrayList<>();**

**planetObstacles.add("(5,5)");**

**planetObstacles.add("(7,8)");**

**MarsRover rover = new MarsRover(10, 12, planetObstacles);**

**// It determines whether, or not, the planet contains an obstacle in a cell.**

**boolean obstacle = rover.planetContainsObstacleAt(7, 8);**

**// It lets the rover move on the planet according to a command string. The // return string contains the new position of the rover, its direction, and the obstacles it has encountered while moving on the planet (if any).**

**String commandString = "f";**

**String returnString = rover.executeCommand(commandString);**

# **User Stories**

Remember to read, implement, and test the user story once at a time (in the provided order). Therefore, do not read the next user story, if the current one is not done.

## **1 - Planet**

The planet, where the rover moves, is represented as a grid with *x* and *y* coordinates. The origin of the grid, namely *(0,0)*, is at the bottom-left corner (see the figure below).

The planet may contain obstacles in its cells. The obstacles are defined through a list of strings (without white spaces), where each string (representing an obstacle) is formatted as follows: *“(oi\_x,oi\_y)”*. For example, a list containing the strings *“(0,1)”* and *“(2,2)”* indicates that there are two obstacles on the planet: the former in the cell with coordinates *(0,1)* and the latter in the cell with coordinates *(2,2)* (see the figure below).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **2** |  |  |  |  | **X** | |
|  | *(0,2)* |  | *(1,2)* |  | *(2,2)* |  |
| **y 1** | | **X** | |  |  |  |  |
| *(0,1)* |  | *(1,1)* |  | *(2,1)* |  |
|  | **0** |  | |  |  |  |  |
|  | *(0,0)* |  | *(1,0)* |  | *(2,0)* |  |
|  |  | **0** | | **1**  **x** | | **2** | |

**Requirement:**

* Implement **MarsRover.planetContainsObstacleAt(int x, int y)** to determine whether, or not, a cell contains an obstacle.

**Example:** A 10x10 planet with two obstacles at *(4,7)* and *(2,3)*.

## **2 - Landing**

The rover has a position (defined by *x* and *y* coordinates) and a direction—i.e., where the rover is facing. The direction can be North (*N*), South (*S*), West (*W*), or East (*E*).

The rover starts its journey from the landing position—i.e., the position with coordinates *(0,0)*—, facing North (see the figure below).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **2** |  |  |  |  |  | |
|  | *(0,2)* |  | *(1,2)* |  | *(2,2)* |  |
| **y 1** | |  | |  |  |  |  |
| *(0,1)* |  | *(1,1)* |  | *(2,1)* |  |
|  | **0** | **⬆** | |  |  |  |  |
|  | *(0,0)* |  | *(1,0)* |  | *(2,0)* |  |
|  |  | **0** | | **1**  **x** | | **2** | |

Immagine che contiene diagramma

Descrizione generata automaticamente

To let the rover move on the planet, the rover receives a command string. Once the rover has executed the commands (specified in the command string), it returns a string containing its new status, together with the obstacles it has encountered (if any). The rover status is a string (without white spaces) formatted as follows: *“(x,y,dir)”*. For example, *“(0,1,W)”* indicates that the rover is in the position with coordinates *(0,1)*, facing West. It is easy to follow that *dir* is equal to: *N* (North), *S* (South), *E* (Est), *or W (West)*.

When the rover receives an empty command string, it returns a string consisting of the landing status—i.e., the string *“(0,0,N)”*.

**Requirement:**

* Implement **MarsRover.MarsRover(int planetX, int planetY, List<String> planetObstacles)** to define a planet (already done in the previous user story) and to initialize the rover to the landing position, facing North.
* Implement **MarsRover.executeCommand(String commandString)** to let the rover return a string consisting of its landing status when it receives an empty command string.

**Example:** When the rover receives an empty command string, it returns a string consisting of its landing status, namely: *“(0,0,N)”*.

## **3 - Turning**

The rover turns right or left when it receives a command string *“r”* or *“l”*, respectively. When the rover turns, it remains on the same grid's cell while it changes its direction.

**Requirement:** Implement **MarsRover.executeCommand(String commandString)** to let the rover turn right or left and then return a string consisting of its new status.

**Example:**

* A rover with status *“(0,0,N)”,* after executing the command string *“r”*, returns a string consisting of its new status, namely: *“(0,0,E)”*.
* On the other hand, a rover with status *“(0,0,N)”*, after executing the command string *“l”*, returns: *“(0,0,W)”*.

*To help you reason about the inputs and expected outputs of your test cases, from here onwards you can use the following grid template:*

* [*https://docs.google.com/spreadsheets/d/1vOM35e6wX1h0Huwmure2miBlC2wh7YswEwVSsOgq7Zc/edit?usp=sharing*](https://docs.google.com/spreadsheets/d/1vOM35e6wX1h0Huwmure2miBlC2wh7YswEwVSsOgq7Zc/edit?usp=sharing)*.*

*Note that you cannot modify the grid template directly. However, you can download it as an Excel file or OpenDocument file (go to “File” > “Scarica”) and then modify it on your PC. Alternatively, you can draw a grid on a piece of paper and then reason about the inputs and expected outputs of your test cases.*

**4 - Moving forward**

The rover moves forward (of one cell) when it receives the command string *“f”*.

**Requirement:** Implement **MarsRover.executeCommand(String commandString)** to let the rover move forward and then return a string consisting of its new status.

**Example:** A rover with status *“(7,6,N)”*, after executing the command string *“f”*, returns a string consisting of its new status, namely: *“(7,7,N)”*.

**5 - Moving backward**

The rover moves backward (of one cell) when it receives the command string *“b”*.

**Requirement:** Implement **MarsRover.executeCommand(String commandString)** to let the rover move backward and then return a string consisting of its new status.

**Example:** A rover with status *“(5,8,E)”*, after executing the command string *“b”*, returns a string consisting of its new status, namely:*“(4,8,E)”*.

**6. Moving combined**

The command string can contain a combination of single commands (i.e., *“r”*, *“l”*, *“b”*, and *“f”*). When the command string corresponds to a combination of single commands, the rover sequentially executes every single command and then returns a string containing its new status.

**Requirement:** Implement **MarsRover.executeCommand(String commandString)** to let the rover execute a sequence of single commands and then return a string consisting of its new status.

**Example:** A rover with status *“(0,0,N)”*, after executing the command string *“ffrff”*, returns a string consisting of its new status, namely: *“(2,2,E)”*.

**7. Wrapping**

The planet is actually a sphere. Therefore, when the rover goes beyond an edge, its new position is at the opposite edge.

**Requirement:** Implement **MarsRover.executeCommand(String commandString)** to let the rover go beyond the edges and then return a string consisting of its new status.

**Example:** On a 10x10 planet, a rover with status *“(0,0,N)”*, after executing the command string *“b”*, returns a string consisting of its new status, namely: *“(0,9,N)”*.

**8. Single obstacle**

The rover can encounter an obstacle while executing commands and thus moving on the planet. When this happens, the rover cannot pass through the obstacle (i.e., it does not move). Therefore, it takes note of the encountered obstacle and then tries to continue executing the remaining commands (in the command string). Once all the commands have been executed, the rover returns a string containing its new status and the encountered obstacle. If the same obstacle has been encountered twice or more, it is reported only once.

**Requirement:** Implement **MarsRover.executeCommand(String commandString)** to let the rover deal with an obstacle and then return a string containing its new status and the encountered obstacle.

**Example:** There is one obstacle on the planet at coordinates *(2,2)*. A rover with status *“(0,0,N)”*, after executing the command string *“ffrfff”*, returns a string containing its new status and the encountered obstacle, namely: *“(1,2,E)(2,2)”*. Note that the same obstacle has been encountered twice but reported only once.

**9. Multiple obstacles**

The rover can encounter multiple obstacles while executing a command and thus moving on the planet. When this happens, the rover (which cannot pass through obstacles) returns a string containing its new status and the encountered obstacle (in the order with which they have been encountered). If the same obstacle has been encountered twice or more, it is reported only once.

**Requirement:** Implement **MarsRover.executeCommand(String commandString)** to let the rover deal with more obstacles and then return a string containing its new status and the encountered obstacles.

**Example:** There are two obstacles on the planet at coordinates *(2,2)* and *(2,1)*. A rover with status *“(0,0,N)”*, after executing the command string *“ffrfffrflf”*, returns a string containing its new status and the encountered obstacles, namely: *“(1,1,E)(2,2)(2,1)”*. Note that the first obstacle has been encountered twice but reported only once.

**10. Wrapping and obstacles**

The rover can encounter an obstacle when trying to go beyond an edge.

**Requirement:** Implement **MarsRover.executeCommand(String commandString)** to let the rover deal with obstacles, when trying to go beyond edges, and then return a string containing its new status and the encountered obstacles.

**Example:** On a 10x10 planet, there is an obstacle at coordinates *(0,9)*. A rover with status *“(0,0,N)”*, after executing the command string *“b”*, returns a string containing its new status and the encountered obstacle, namely: *“(0,0,N)(0,9)”*.

**11. A tour around the planet**

The rover takes a tour around the planet, encountering several obstacles and going beyond edges in any dimension.

**Requirement:** Implement **MarsRover.executeCommand(String commandString)** to let the rover take a tour around the planet and then return a string containing its new status and the encountered obstacles.

**Example:** On a 6x6 planet, there are obstacles at coordinates *(2,2)*, *(0,5)*, and *(5,0)*. A rover with status *“(0,0,N)”*, after executing the command string *“ffrfffrbbblllfrfrbbl”*, returns a string containing its new status and the encountered obstacles, namely: *“(0,0,N)(2,2)(0,5)(5,0)”*.