COMP20050 - Group 47 - Features
Sprint 2

Student Numbers: 22441636, 22450456, 22715709

In this sprint our main goal was to begin the ray development and start implementing the rays path through the board for the no absorption case, absorption case and 60 degree reflection case. We also wanted to also build on our previous sprint of letting the setter place atoms and now also let the experimenter to shoot rays from various positions.

Implemented Features:

Ray:

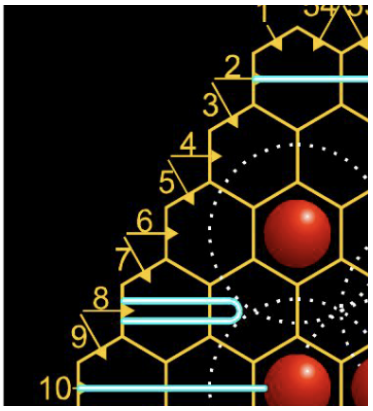
Our ray implementation consists of a ray class which holds its x-coordinate, y-coordinate, orientation which are the main instance variables. The way our ray travels through the board is through manipulating its current coordinates with class methods which move the ray depending on its orientation. A ray can have 6 different orientations, 0, 60, 120, 180, 240 and 300 and depending on the orientation, it will traverse the board differently.

RayInputMap:

As the core game allows users to input a ray from 1 - 54, we had to come up with a way to effectively map each individual input to a corresponding coordinate and orientation in which we would initialise the ray's instance variables to. For this, we decided to use a hashmap data structure in our board class, which efficiently maps 1 - 54 with another object called a RayInputMap which holds just an x coordinate, y coordinate and orientation.

```
//mapping data structure for ray inputs
public final HashMap<Integer, RayInputMap> inputMapping = new HashMap<>();
```

While the use of a hashmap not only allows the board to access a mapping in efficient, constant time, it also allows us to get used to using effective data structures in real scenarios. The map is initialised through a function which runs around the outside edge of our board getting all the coordinates and then working out the individual orientations through patterns which emerge.



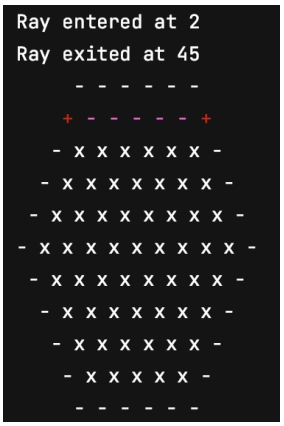
Example here of all even numbers from 1 - 10 being given an orientation we call “0” and all odd numbers being given an orientation we call “300”.

As our board is extended with an extra outer rim we can access to allow for ray marker placement, it is important to map the correct orientation to each hexagon as in this image, a ray sent from 2 and 3 will have the same starting hexagon but their orientations are different.

Ray Paths:

As we have previously mapped each part of a circle of influence to an orientation of type int, it makes calculating a ray's fate easier as in certain instances we can use maths to find out what the fate of a ray should be. As a ray moves through our board, at each position, provided that it will not overwrite anything a ray graphic is placed to denote a rays path. This continues until the ray exits and ultimately shows the ray's entire path.

The case in which a ray encounters nothing on its journey through the board in a sense implemented itself as our loop in which a ray moves will terminate as soon as the edge of the board is encountered.



This image shows a ray simply going through the board and exiting without encountering any obstacles.

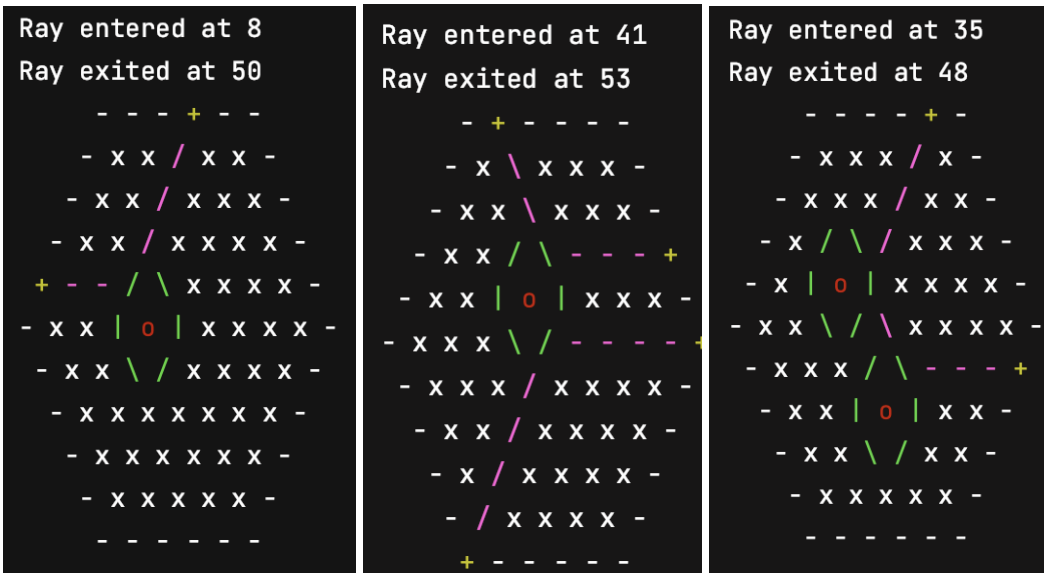
We next wanted to implement the case in which a ray encounters an atom. This was calculated using the rays current orientation compared to the orientation of the circle of influence it was encountering. We found that for our system, the following equation would successfully calculate an absorption from all 6 possible directions.

```
if(orientation - c.getOrientation() == 90 || orientation - c.getOrientation() == -90
        || orientation + c.getOrientation() == 360){
```

Orientation being the rays orientations and c.getOrientation() corresponding to the encountered circle of influences orientation.

This image shows the case in which a ray enters but does not exit deducing absorption.

Finally we implemented the case in which a ray will get deflected 60 degrees from all possible directions. Again we used certain mathematical equations to find out what the rays' new orientation should be. As there are various different combinations in which a ray can hit a circle of influence we made sure that all situations were accounted for.



These are some examples of our ray exerting 60 degree reflections.
In sending a ray, a ray marker gets placed at where the ray enters and then exits and depending on what it does/does not encounter on the board, it receives a different colour. Red: no encounters, Green: absorption, Yellow: 60 degree reflection

View:

In sticking with our MVC architecture, we updated our printBoard() method to also print previously placed “ray graphics” which can be seen in the above images; however a ray graphic is only placed onto the board provided it will not overwrite an important object on the board. Ray graphics will however overwrite themselves but this will be further developed.

PlayerInput:

With regards to the controller side of our MVC architecture, we have created a new class method which takes in an experimenter's input for the ray but it can also recognise when the player wishes to stop sending rays through hitting the “enter” key on the keyboard. The class also now has a class HashSet data structure which stores all previously sent rays. Again in this instance it made sense to use this efficient data structure to hold previously entered ray locations due to its constant look up times and also gave us a chance to use this effective data structure in a real scenario.

Testing:

As our ray class has an entrance and exit point, it made it significantly easier in testing all the possible cases for ray behaviour.

We first of all tested the ray behaviour whereby no atom was detected. This was done by placing a series of atoms on the board and then sending rays such that they didn't encounter any of these atoms.

```
Ray ray = b.getSentRays().get(0);
assertEquals(ray.getInput(), actual: 2);
assertEquals(ray.getOutput(), actual: 45);
```

Next we tested for where the ray was absorbed by the atom. Again, atoms were placed on the board and rays sent such that they should be absorbed by the atom. In this case, where a ray is absorbed, the output is -1.

```
Ray sentRay = b.getSentRays().get(0);
assertEquals(sentRay.getInput(), actual: 2);
assertEquals(sentRay.getOutput(), actual: -1);
```

We tested whereby the ray hits the atom at a 60 degree angle. The expected output was tested for as well as the deflection type.

```
Ray ray = b.getSentRays().get(0);
assertEquals(ray.getInput(), actual: 32);
assertEquals(ray.getOutput(), actual: 44);
assertEquals(ray.getDeflectionType(), actual: 60);
```

We then tested that the correct ray marker colour was displayed on the board for scenarios where the ray was absorbed, deflected or simply passed through the board.

```
b.sendRay( input: 2);
String red = "\u001B[31m";
RayMarker r = (RayMarker) b.getBoardPosition( x: 4, y: 1);
assertEquals(red,r.getColour());
```

Finally we tested the user input and ensured that accurate ray entrance point is inputted. As this is more part of the UX, we decided not to throw an error if invalid entrance points were input but instead ask a user to try again. We tested this to make sure that for ray input, only valid numbers or a new line character were inputted.

This concluded our tests for this sprint which allowed us to see that our program, so far, is performing as expected allowing us to conclude sprint 2 as fully completed.