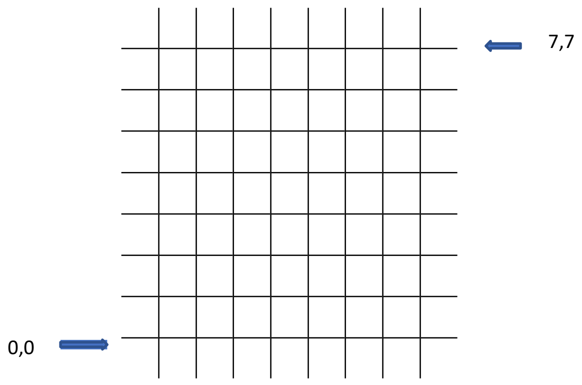
Problem Description:
A city is laid out as a rectangular grid as you see in the figure below, where the first intersection at bottom left is identified as (0,0) and the top right intersection as (n-1, m-1). In this figure the number of the east-west and north-south streets is 8 (i.e. n = 8, m = 8).
A self driving car, which is left at a random intersection, is given the instruction to drive to a particular intersection. The car should be able to find the shortest path from the source intersection to the destination intersection. It is assumed that intersections have an equal distance from each other.



One of the basic strategies that it uses, is to choose a path that intuitively seems correct. With this strategy, if the destination is on the west, a path that goes to the east should not be chosen. If the destination is on the south, the path that goes to the north should not be chosen. For example, if the car is at intersection (3, 3) and the destination is (10,10), it seems reasonable that the car chooses a path to the north, where the rows > 3 and then to the east where the column > 3. It is also possible to go to the east first and then to the north to get to the destination.

Task 1:
The first task is for you to construct the map of the city by implementing the constructor of the class. As you will see from the code, the constructor gets the number of rows and columns to make the map of the city.

```
        public Map(int , int)
```

Task 2:
The second job for you is to write a **recursive** method that returns a path from a source to a destination. Implement the following method, when we know the car is traveling to south west. The starting and stopping intersection is given as the input parameter. You can see the description of the input parameters in the JavaDoc of this method. Please note that the following method is a 'private' method that cannot be tested by the tester directly. The tester can only test this method indirectly by the method that is implemented for task 4. My suggestion is to partially implement task 4, so the tester can test this method.

```
        private String goSouthWest (int , int , int , int , String )
```

```
sample input: 5,5, 4, 1, " "
one sample Expected output: (4,5) (4,4) (4,3) (4,2) (4,1)
```

Task 3:
If you were able to complete the second task, this task is a piece of cake for you. Use the same approach that you used for task 2, to implement the three following methods.

```
        private String goSouthEast (int, int, int, int, String)
        private String goNorthEast (int, int, int, int, String)
        private String goNorthWest (int, int, int, int, String)
```

Again, the above methods are private, which cannot be tested by the tester directly. It's better to partially implement task 4, in order to test these methods.

Task 4:
As you probably noticed, all the 4 methods above are private. It means a client (i.e. a self driving car) cannot use these methods. So as your fourth task, you need to implement *getPath* method, which has two responsibilities. The first responsibility is to check if all the inputs are valid. If it was not valid, then this method should issue an *IllegalArgumentException.* It is your job to figure out what a valid input means.

The second job of this function is to decide in which direction the car should travel. It should call one of the four methods that you have just implemented, depending on the direction that the car is moving. This method is not a recursive method.

```
    public String getPath (int, int, int, int, String path)
```

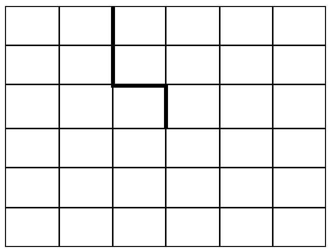
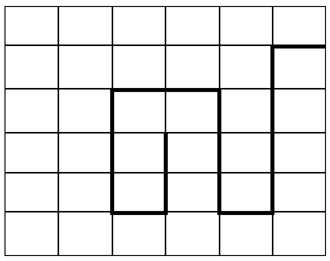
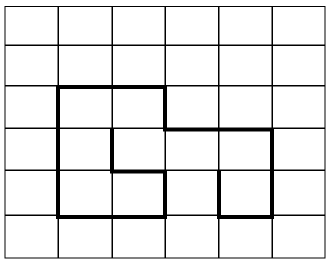
Task 5:
Suppose that a police car is chasing the self-deriving car. The car wants to get out of the city map (i.e. the grid). We know that our police are capable enough to catch the car ultimately, so we let the self-driving car to be adventurous for a while before the police car stops it. Whenever the car passes an intersection, a police car is deployed at that intersection. This means that the car cannot pass an intersection for the second time, otherwise it will be caught by the police. So, as long as the car does not pass an intersection more than once, it can travel to any direction to finally get out of the city. Please see figure below that shows some example of the path that our car has passed in the different experiments. The starting point for first two examples is intersection (3, 3) and for the third example is (3, 2). In the two first example, the car was able to exit the city, however in the third example, it was caught by the police.

As the car starts to travel, it chooses the next intersection randomly. That is why the car may get caught by the police. If the car is caught by the police, the game starts over. The car is positioned at its original place, where it starts to randomly choose a path out of the city.

Your job for this problem is to implement the following method that takes in the intersection number and produces a path out of the city. If the car was caught by the police, the method should start over and keep doing this until it finds a path out.

Please note that we are looking for a **recursive** solution, and you are welcome to use indirect recursion if you needed to. This means that you can use private helper methods to be used with the following method to solve the problem.

```
    public String findPath (int, int)
```




The output of this method is a path to the outside of the map. For example, for the first example above, the output should be:
(3,3) (4,3) (4, 2) (5,2) (6, 2)
Please note that the starting point, in this example (3,3), must be included.

What to Submit:
you only submit one file, which is the completed Map.java file.

Marking Scheme:

Please note that the submitted code, should be compiler-error free. In case the code has compiler error, no mark will be awarded.

60% on the correctness of the code. This includes:

- 10% on the correctness of task 2. You will only receive the points if you implement it recursively.
- 6% on the correctness of task 3. You only receive the points if you implement them recursively.
- 4% on the correctness of task 4.
- 40% on the correctness of task 5. You only receive the points if you implement them recursively.

40% on code style. We assume that you get this 40% and then we start to deduct marks for the following problems.

- -2 for every complex code that does not contain a comment. The maximum deduction is 10 points.
- -2 for every meaningless identifier name. The maximum deduction is 10 points.
- -2 for every indentation flaw. The maximum deduction is 10 points.
- -2 for not correctly implementing the constructor.
- -2 points for unnecessary codes. The maximum deduction is 10 points.