2. Treasure Hunt

In a treasure hunt game, multiple players search on the same map for hidden treasures. Players could move up, down, left and right for multiple steps. If a player steps into a treasure on the route, s/he acquires its score, and the treasure is removed from the map.

The game ends when all the treasures are found. Now we can check the winner, the one who got the highest score with the least steps taken.

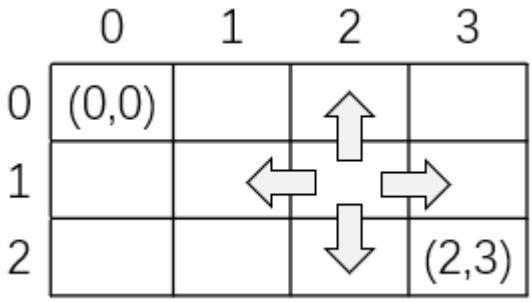
Now let's develop this game by implementing the following classes.

2.1 Position

Fields

```
private int row;
private int col;
```

This class is used to indicate a player's or a treasure's position (`row`,`col`) in the map.
A map can be viewed as a 2D table. Each grid in a table can be located by its row number and column number.
Assume that rows and columns start with 0. So the grid at the top left corner of the map has a position of (0,0).



Constructors

```
public Position(int row, int col)
```

Methods

```
public boolean equals(Object pos)
```

Write the `equals` method so that if the position of `this` and `pos` are the same, return `true`, or else return `false`.

Note that you need to explicitly cast `pos` to the `Position` type before making a comparison, i.e., `Position otherPos = (Position) pos`.

Also provide `getter` and `setter` methods for `row` and `col`.

2.2 Treasure

Fields

```
private final int score;
private final Position pos;
```

Constructors

```
public Treasure(int score, Position pos)
```

Create a treasure whose score is specified by `score` and position is specified by `pos`.
We can assume that `score>0`.

Once a treasure has been created, its score and position can no longer be modified.

Methods

There should be `getter` methods for `score` and `pos`.

2.3 Map

Fields

```
private final int rows;
private final int columns;
private boolean isActive;
private Treasure[] treasures;
```

Constructors

```
public Map(int rows, int columns, Treasure[] treasures)
```

Initialize a map with number of rows and columns
specified by `rows` and `columns`, and place `treasures` on the map.
We can assume that the rows and columns of the map and the positions of the treasures are all valid.
In addition, *one position can only have one treasure*.

After initialization, the map should be active,
and its size can no longer be changed.

Methods

```
public int hasTreasure(Position pos)
```

Return the treasure's score if the given `pos` has a treasure; return 0 otherwise.

```
public void update(Position pos)
```

If the treasure on `pos` has been found by players, the map should be updated to remove this treasure. The map is no longer active as soon as all teasures have been found.

```
public boolean isActive()
```

Return `true` if the map is active, `false` otherwise.

2.4 Player

Fields

```
private final int id;
private int score;
private int steps;
private Position pos;
private Map map;
```

- `id`: a unique ID for each player. Different players should have different IDs.
- `score`: the current scores of the player. Initialized to 0.
- `steps`: the number of steps taken by the player. Initialized to 0.
- `pos`: the current position of the player. Multiple players can be at the same position at the same time.
- `map`: the current map.

Constructors

```
public Player(Map map, Position initialPos)
public Player(Map map, Position initialPos, int maxStepAllowed)
```

Initialize a player, by specifying his/her map and initial position. *We can assume that players' initial positions do not have treasures*.

We could also specify the maximum number of steps allowed for the player, meaning that the player can no longer move if s/he has already taken `maxStepAllowed` of steps. If `maxStepAllowed` is not specified, then the player can take as many steps as s/he wants when the map is active.

Methods

First, please provide public `getter` methods for `id`, `score`, `steps`, and `pos`.

```
public boolean move(Direction direction, int steps)
```

The player can move to 4 directions: up, down, left, and right. Please define a **enum** type `Direction`, with four values `UP`, `DOWN`, `LEFT`, and `RIGHT` for this purpose.

We should also specify the number of `steps` to move.

The method returns `true` if the player can successfully move the given steps to the given direction. The method returns `false` if:

- The map is **inactive**, meaning that all treasures have been found. In this case, players can no longer move.
- The player cannot move at all because s/he will hit the boundary of the map.
- The player cannot move because s/he has exceeded the `maxStepAllowed`.
- The player can move. However, s/he cannot finish all the steps because s/he will hit the boundary during the process. For example, if `steps` is 5, but the player can only take 2 steps before hitting the boundary. Then the player will take only 2 steps and stop.

```
public boolean equals(Object player)
```

Write the `equals` method so that if the `id` of `this` player and the `id` of the given `player` are the same, return `true`, or else return `false`.

2.5 GameSystem

Fields

```
private List<Player> players;
```

The `players` field maintains a list of players for the current game.

Methods

```
public void addPlayer(Player player)
```

Add a new player to this game.

```
public Map newGame(int rows, int columns, Treasure... treasures)
```

Start a new game, by creating a map of the given `rows` and `columns`, and `treasures` on the map. Return the created map.

```
public Player getWinner()
```

Return the player with the highest score as the winner. If two players have the same score, return the player with the least steps.

We can assume that this method is invoked only after all treasures have been found, and there is only one winner.

Fields (and other methods)

In all of these classes, you could define any fields and other methods as you want to help you implement the requirements.
Our tests will only invoke the required methods to evaluate the correctness of your code.

Test the Game

Here we have a sample code for testing the game. The code works on a map as shown below. Please read the code to figure out how each player moves.

	0	1	2	3	4
0	player1	5	10	player3	2
1				15	player2
2					

```
public class Test {
    public static void main(String[] args) {
        GameSystem game = new GameSystem();

        Treasure[] treasures = new Treasure[4];
        treasures[0] = new Treasure(5, new Position(0,1));
        treasures[1] = new Treasure(10, new Position(0,2));
        treasures[2] = new Treasure(2, new Position(0,4));
        treasures[3] = new Treasure(15, new Position(1,3));
```

```
        Map map = game.newGame(3,5, treasures);

        Player player1 = new Player(map, new Position(0,0));
        Player player2 = new Player(map, new Position(1,4), 4);
        Player player3 = new Player(map, new Position(0,3));

        game.addPlayer(player1);
        game.addPlayer(player2);
        game.addPlayer(player3);

        player1.move(Direction.DOWN, 2);
        player1.move(Direction.RIGHT, 2);
        player1.move(Direction.UP, 4); //player1 can only move 2 steps UP before
hitting the boundary

        player2.move(Direction.LEFT, 2);
        player2.move(Direction.UP,1); //player2 cannot get the treasure at (0,2),
which is already obtained by player1

        player1.move(Direction.LEFT, 3); //player1 can only move 2 steps LEFT
before hitting the boundary

        player2.move(Direction.RIGHT, 2); //player2 can only move 1 step RIGHT
before reaching the maximum steps allowed

        player3.move(Direction.RIGHT,1);
        player3.move(Direction.DOWN, 2); //player3 cannot move because all
treasures have been found and the map is inactive

        assert(player1.getSteps() == 8);
        assert(player1.getPosition().equals(new Position(0,0)));
        assert(player1.getScore() == 15);

        assert(player2.getSteps() == 4);
        assert(player2.getPosition().equals(new Position(0,3)));
        assert(player2.getScore() == 15);

        assert(player3.getSteps() == 1);
        assert(player3.getPosition().equals(new Position(0,4)));
        assert(player3.getScore() == 2);

        Player winner = game.getWinner();
        assert(winner.equals(player2)); //player2 and player1 have the same
scores, but player2 takes less steps, so player2 is the winner
    }
}
```