Due Date: March 25, 2022 at 11:59 PM CDT

The Notorious Mouse Named Suzie

Instructions

- You must complete the "**Blanket Honesty Declaration**" checklist on the course website before you can submit any assignment. You will not be able to see the dropbox if you have not accepted the Honesty Declaration.
- Only submit the **java files**. Do ***not*** submit any other files, unless otherwise instructed.
- To submit the assignment, upload the specified files to the **Assignment 2** folder on the course website.
- Assignments must follow the **programming standards** document published on UMLearn.
- After the due date and time, assignments may be submitted but will be subject to a late penalty. Please see the ROASS document published on UMLearn for the course policy for late submissions.
- If you make multiple submissions, only the **most recent version** will be marked.
- These assignments are your chance to learn the material for the exams. Code your assignments independently. We use software to compare all submitted assignments to each other, and **pursue academic dishonesty vigorously**.
- Your Java programs must compile and run upon download, without requiring any modifications.

Suzie in the Big City

Suzie is a mouse in the big city. She needs to collect a lot of cheese that will last her throughout the cold winter months. Each day, Suzie travels to a new city to gather up all the cheese crumbs she can find.

Cheese crumbs are gathered in boxes, so Suzie has to open each box to steal the cheese.

News of the cheese prowler has spread across neigbouring cities like wildfire. Concerned cities determined to protect their precious cheese have hired exterminators to take out the notorious Suzie. So she has to be extra careful now. Unbeknown to Suzie, exterminators now hide "traps" in some of these boxes, to trap Suzie when she opens it.

Your objective

Your goal is to lead Suzie to the cheese in these dangerous times.

Phase 1: The BigCity

A city is defined by an MxN grid (M may not be equal to N). The city will show the starting position of Suzie (denoted by *s* below). **Suzie's starting position is always at (0, 0)**.

The city will also show the boxes (denoted by **'b'**).

Valid "paths" through the city are denoted by the **'.'** character.

The following is a 4x5 city grid (4 rows and 5 columns), with 3 boxes which may contain cheese **or a trap!**

```
s . . b .
. . . . .
. . . . .
. b . . b
```

First create the city grid, named class **BigCity**.

The **BigCity** class should have:

- The following instance variables:
  - a 2D array of **char** to hold the grid
  - an **int** to hold the number of boxes in the grid
  - an **int** to hold the number of cheese in the boxes. *This value is less than or equal to the number of boxes in the grid*
  - a 2D array of **int** to hold the positions of the boxes that contain cheese
- A constructor **BigCity(int rows, int cols, int numBoxes, int numCheese, int [ ][ ] cheesePositions)** which creates a grid. It assigns the 2D array instance of cheese positions to the **cheesePositions** parameter passed.
- A **private** method **fillGrid()** that:
  - marks Suzie's start position at (0, 0) with 's'
  - marks the positions in **cheesePositions** as 'b'.
  - marks (numBoxes **minus** numCheese) **random** positions in the grid as 'b'. **It cannot mark (0, 0) or any positions in cheesePositions**. You can create a private method to do this.
  - marks the remaining positions as '.'

  This method should be called from the constructor you defined above.
- A standard **toString** method, which returns a text representation of the city.

Given the grid above, the constructor may be passed the following:

- rows = 4
- cols = 5
- numBoxes = 3
- numCheese = 2 *(also could be 3, 1, or 0 but not greater than 3)*
- cheesePositions = {{3, 1}, {0, 3}} *(the number of rows is always **numCheese**, the number of of columns is always 2. Each row is the coordinates of a piece of cheese.)*

Given this sample input, you can tell that the box at (3, 4) contains a trap for the grid shown above, as it is not in the list of `cheesePositions` provided.

Phase 2: Loading BigCity from files

We want to be able to load a city from a file. The format is as follows:

- First line - The number of rows and the number of columns of the city grid
- The next (# rows) lines - The city grid separated by whitespace
- The following line - The number of boxes
- Next line - The number of cheese
- Next (numCheese) lines - The positions of cheese separated by a space in the format: row column

For example:

```
6 5
s . . . .
. . b . .
. . . . .
b b . . .
. . . . b
. . . . .
4
2
3 0
1 2
```

is a valid file with a 6x5 city grid with 4 boxes and 2 cheese at locations (3, 0) and (1, 2).

To do this, add the following to your BigCity class:

- Add a second constructor **BigCity(String fileName)**. This constructor reads the data in the text file specified by "fileName". **Do not forget to initialize your instance variables**.

  This method should throw an **IOException** with an appropriate message if the data is not valid. Special exception messages include:
  - File specified was not found
  - Bad dimensions for the grid
  - Inaccurate number of rows of data to read
  - Inaccurate number of columns of data to read

  Your error messages should have as much detail as possible. See the sample output for what is expected.

Sample output 1

Input file:

```
s .
. b
1
1
1 1
```

Exception Thrown: java.io.IOException: No dimension to read

Sample output 2

Input file:

```
2 2
s .
. b
1
1
```

Exception Thrown: java.io.IOException: Inaccurate number of rows of cheese positions in the file. Saw 0 expected 1

Sample output 3

Input file:

```
4 5
s . . .
. b . .
. . . .
1
1
1 1
```

Exception Thrown: java.io.IOException: Inaccurate number of columns of cheese in the file. Saw 4 expected 5

Do not bother error-checking the individual characters on the grid - the only thing you should check is if the number of rows and columns match what was provided. If they do, assume that the grid is "good".

Note: you do not need to do all the error handling in this phase to start working on the next phase.

Phase 3: Moving through BigCity

Add the following to enable moving along the city grid.

Create 2 **count** variables: one to track how many cheese crumbs Suzie has collected, and one to track how many moves Suzie has made moving through the grid. *Do not forget to update both constructors to initialize them*.

Create 1 **boolean** instance variable to indicate that Suzie is still loose and roaming the city.

Create a method **void move(char direction)**. This will move Suzie in the direction specified. The following characters will represent the movements/direction along the grid

- **'w'** - up
- **'s'** - down
- **'a'** - left
- **'d'** - right

Throw appropriate **IndexOutOfBoundsException** for moving off the grid. If the move is valid, then process the move using the following helper method.

Create method **private void processMove(char direction)**, which determines the next outcome of the game based on the direction from the **move()** method.

- A move onto a valid path ('.') increases the number of moves Suzie has made.
- A move onto a 'b' can either:
  - increase Suzie's cheese collection by 1 (as well as her move count), or
  - end the game (and increase her move count) by calling the **endTerror()** method you will implement in the next phase.

Update the grid to show Suzie's movements. If she moves, replace her last position in the grid with '**.**' and set the char at the new position to '**s**'. If Suzie collects a cheese, replace the '**b**' with '**s**' to show where she currently is. When she moves from that location, update the grid accordingly.

You may consider to create a method **isBoxCheese(int row, int col)** to check if the position (row, col) is one of the elements of the array **cheesePositions**. If yes, **isBoxCheese** returns true, otherwise, it returns false.

Phase 4: Game ending condition

Suzie has to keep scouring the city for boxes of cheese crumbs. She only stops when **any** of these happens:

- Suzie has collected all the cheese in the city, or
- Suzie is caught by a trap in a box.

To do this:

Create method **private void endTerror()**, that changes the value of the variable indicating that Suzie is still roaming the city.

Create a method **boolean isRoamingCity()** that tells the outside world if Suzie is still on the loose in the Big City!

Update the **processMove** method to check if the game is over, and calling **endTerror** to end the game.

Phase 5: Extracting data

First, create your own Exception type named **DataDoesNotExistException**. For this class, implement only a constructor that receives a **String message**. See a short tutorial about how to create a customized Exception here: https://www.baeldung.com/java-new-custom-exception. To implement **DataDoesNotExistException**, you likely need only 5 lines of simple codes.

Then, add two methods to your **BigCity** class:

- **char [ ] extractRow(int rowNum)** that extracts the row specified by the passed integer. 0 will fetch the first row, 1 the second (index-by-zero rules).
- **char [ ] extractColumn(int colNum)**, which extracts a single column, much like extract row.

Both methods should throw a **DataDoesNotExistException** when passed a row or column number that does not exist in your city grid.

Sample output

For example, if your city is a 3x3 grid and the extractRow() method is passed a -1 as input, the output will be:

Exception Thrown: DataDoesNotExistException: BigCity grid does not have a row index of -1

Phase 6: Update the toString with Suzie's stats!

When Suzie's reign of terror is done in a Big City, update the output of your **toString()** method to include one of the following **on a new line**. Optionally watch out for singular and plural tenses.

Sample output:

Suzie outsmarted the exterminators, making 5 moves and collecting all 2 cheese crumbs. She sniffed out the 3 traps.

Suzie outsmarted the exterminators, making 1 move and collecting all 1 cheese crumb. She sniffed out the 1 trap.

Suzie's reign of terror came to an end abruptly after just 10 moves. She was captured with 0 cheese crumbs on her person.

Suzie's reign of terror came to an end abruptly after just 2 moves. She was captured with 1 cheese crumb on her person.

Suzie's reign of terror came to an end abruptly after just 1 move. She was captured with 0 cheese crumbs on her person.

If the game is not over, print nothing.

Hint: You will know if Suzie was captured if the she is no longer roaming the city and somehow there are still cheese crumbs left.

Phase 7: Suzie's Guardian Angel, Acqueline

Suzie's guardian angel, Acqueline, is always watching out for her; she sees her every move and keeps track of them. Acqueline can only keep track of **the last 5 moves Suzie makes**. This enables Suzie go back in time a maximum of 5 steps at once.

To bring Acqueline to life, implement the following:

- Add a partially-filled array to the class that holds a maximum of **5** 2D array of **char**. *Again, remember to initialize it in both contructors*.
  - You do not have to set the dimension of the inner 2D arrays.
- Create a method **private void saveGrid()** that stores a **copy** of the city grid **at index 0** of this partially-filled array.

  If there are elements in the partially-filled array, move them over to the right.

  If the array is full and you need to insert at index 0, discard the last element in the array by moving the rest over to make room at index 0.

  For example: consider adding values a, b, c, d, e, f to a char array in the same manner

```
Adding a: [a, , , , ] size: 1
Adding b: [b,a, , , ] size: 2
Adding c: [c,b,a, , ] size: 3
Adding d: [d,c,b,a, ] size: 4
Adding e: [e,d,c,b,a] size: 5
Adding f: [f,e,d,c,b] size: 5
```

- Modify the **processMove()** method to call **saveGrid()** before modifying the contents of the grid.

- Now, create a method **void undo()**, which replaces the 2D grid instance with the 2D array at index 0 of this partially filled array.

  If the partially-filled array is empty, it should do nothing.

  When this is done, element 0 of the partially-filled array is removed, and the remaining 2D arrays must shuffle left to fill the space.

```
Starting array: [f,e,d,c,b] size: 5
After undo: [e,d,c,b, ] size: 4
```

Hand in

Submit your one Java file **(`BigCity.java`)**. **Do not submit .class or .java~ files!** You do not need to submit the **TestPhaseN.java** files that were given to you. If you did not complete all phases of the assignment, use the UMLearn Comments field when you hand in the assignment to tell the marker which phases were completed, so that only the appropriate tests can be run. For example, if you say that you completed Phases 1-2, then the marker will compile your files with appropriate test cases. If it fails to compile and run, you will lose **all** of the marks for the test runs. The marker will ***not*** try to run anything else, and will not edit your files in any way. **Make sure none of your files specify a package at the top!**