CS102 – Algorithms and Programming II
Programming Assignment 5
Spring 2025

ATTENTION:

- Compress all of the Java program source files (.java) files into a single zip file.
- The name of the zip file should follow the below convention, where you replace the variables "Sec1", "Surname" and "Name" with your actual section, surname and name:

  CS102_Sec1_Asgn5_Surname_Name.zip
- Upload the above zip file to Moodle by the deadline (if not significant points will be taken off). You will get a chance to update and improve your solution by consulting to the TAs and tutors during the lab. You have to make the preliminary submission before the lab day. After the TA checks your work in the lab, you will make your final submission. Even if your code does not change in between these two versions, you should make two submissions for each assignment.

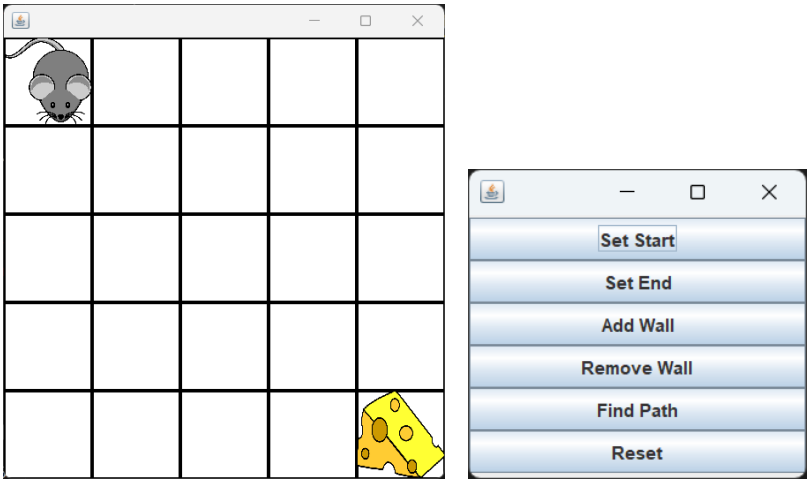
GRADING WARNING:

- Please read the grading criteria provided on Moodle. The work must be done individually. Code sharing is strictly forbidden. We are using sophisticated tools to check the code similarities. The Honor Code specifies what you can and cannot do. Breaking the rules will result in disciplinary action.

Mouse Maze Recursion

For this assignment, you will implement a GUI application where we can build a maze for our mouse to find its way to the cheese. The user can position the start (mouse) and end (cheese) cells in a five-by-five square grid. Users can add or remove walls except for the starting and ending locations. When we click the "Find Path" button, the program should find one of the shortest paths for the mouse to reach the cheese **using recursion.** You will highlight the cells the mouse needs to follow to get to the cheese. If there is no such path, you will display a message about the algorithm's inability to find a valid path. The walls are impassable, and only vertical and horizontal movement is allowed (the mouse cannot move diagonally). You will use images to represent the mouse, cheese, and walls. You may use the images we include with the assignment; you are also free to use any other image you have.

The program should use two frames, one for the maze and the other for the control buttons. The frames should keep a reference to each other so that you can influence the other frame by interacting with one of them. For example, when we click on the "Clear" button of the side frame, the maze frame should update, which requires access to the maze frame from the side frame. By default, the mouse is at the top left corner, and the cheese is at the bottom right corner.



The side frame should use the GridLayout to keep the buttons in one column. The maze frame should include a panel whose paintComponent method is overridden to draw the shapes. Since you need to draw images representing objects, you should first load the image files. You can store the images as BufferedImage objects, which you can read using ImageIO:

```
BufferedImage img = ImageIO.read(new File("filename.png"));
```

Note that the file should be included in your project folder so you can read it successfully. Since reading a file may trigger exceptions, you must surround it with a try-catch block or declare “throws Exception” in the method signature. For example, you may read your BufferedImages using the following code segment:

```
// class variables
BufferedImage mouse;
BufferedImage cheese;
BufferedImage wall;

// inside some method
try {
   mouse = ImageIO.read(new File("mouse.png"));
   cheese = ImageIO.read(new File("cheese.png"));
   wall = ImageIO.read(new File("wall.png"));
} catch (IOException e) {
   System.out.println(e);
}
```

The catch part will catch the exception and print out the exception message in case the read methods fail. The drawImage method of the Graphics object can draw the BufferedImage on screen. Suppose we override the paintComponent method of our panel using the following code:

```
@Override
protected void paintComponent(Graphics g) {
   g.drawImage(img, 0, 0, null);
}
```

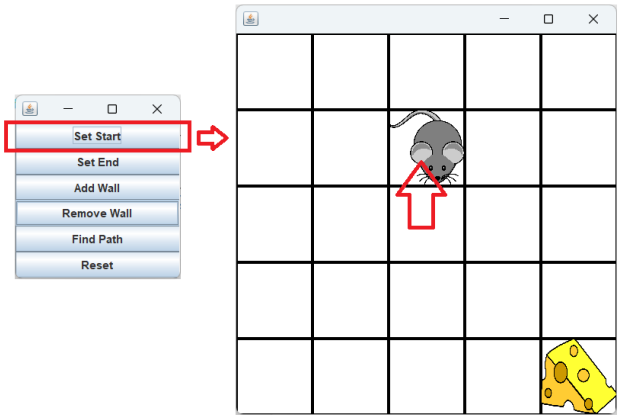
This draws the image in the top left corner using the original size of the image. If we want to draw the image on a different scale, we can use the following version of the drawImage method:

```
// drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)

g.drawImage(img, 0, 0, 40, 40, null); // draw the image as 40 by 40
```

ImageObserver is an interface with methods for handling notification of the state of image loading. For example, if the image we draw is loaded using the internet, we can update the displayed image as new data is available using an ImageObserver. Since we load the images beforehand and only use local data, we do not need to pass an observer. That is why we use null as the observer parameter.

Choosing the “Set Start” option and clicking on one of the cells should update the starting position. For this task, your panel needs to implement MouseListener so that you can handle the mouse events accordingly.

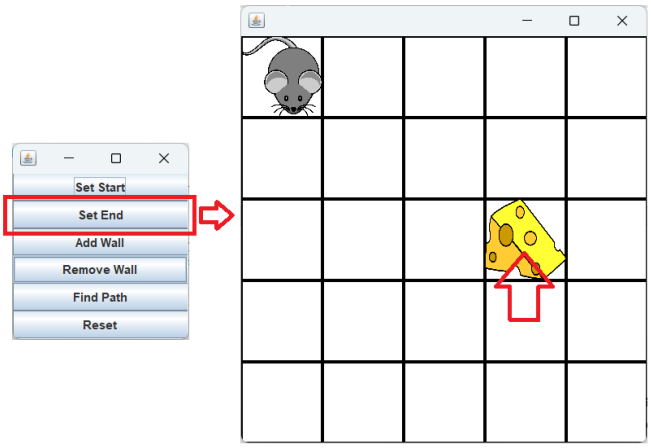


When we click on the panel, you can get the x and y coordinates of the mouse position using the following code:

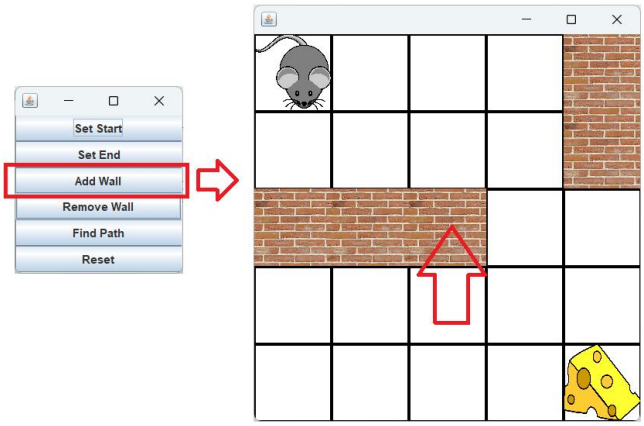
```
@Override
public void mousePressed(MouseEvent e) {
   int x = e.getX();
   int y = e.getY();
}
```

Then, you should do some math to locate the cell to which this mouse coordinates correspond.

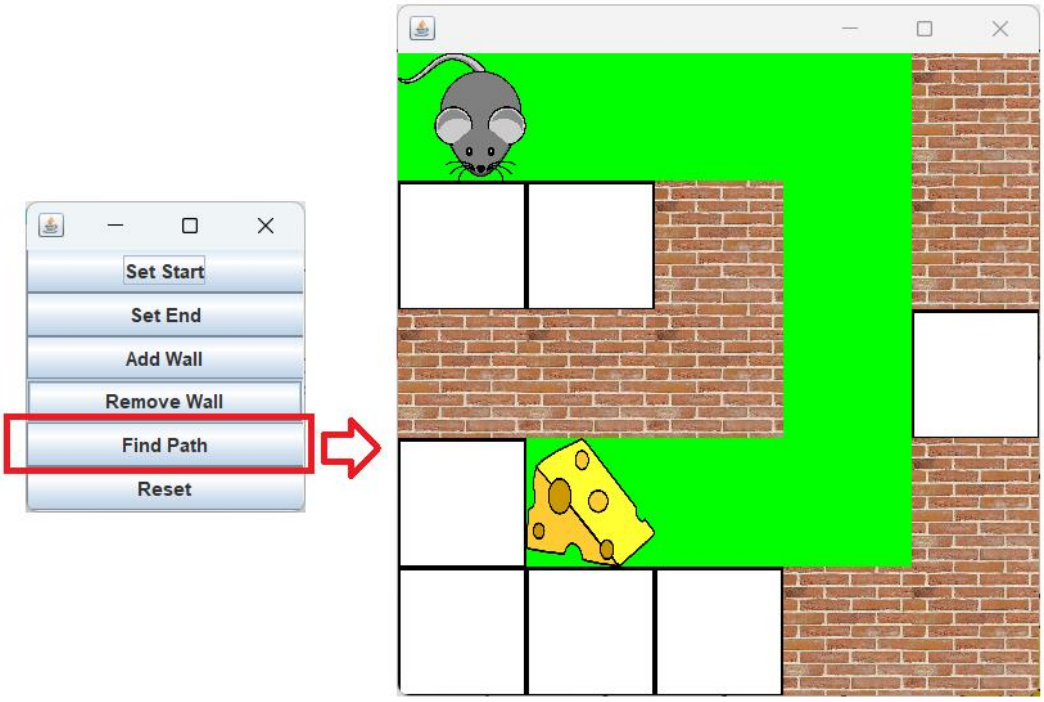
Similarly, the “Set End” button updates the cheese position based on where we click on the maze frame:



When we click the “Add Wall” button, we enter a drawing mode, where clicking on any cell (except for the current starting and ending cells) adds a wall to that cell. We can add multiple walls in this mode. The “Remove Wall” button switches into an eraser, where clicking on any existing wall removes it.



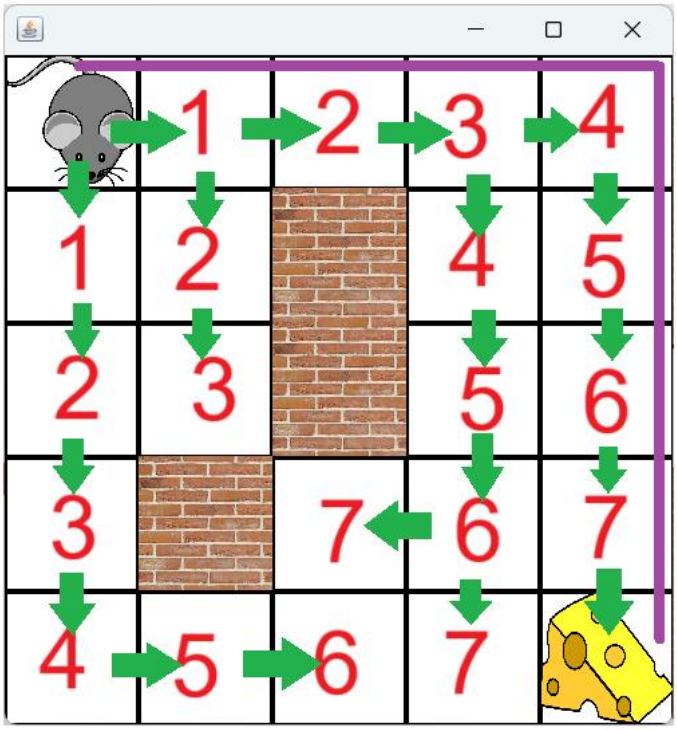
After creating a maze and clicking the “Find Path” button, you should display one of the shortest paths to the cheese on the maze screen.



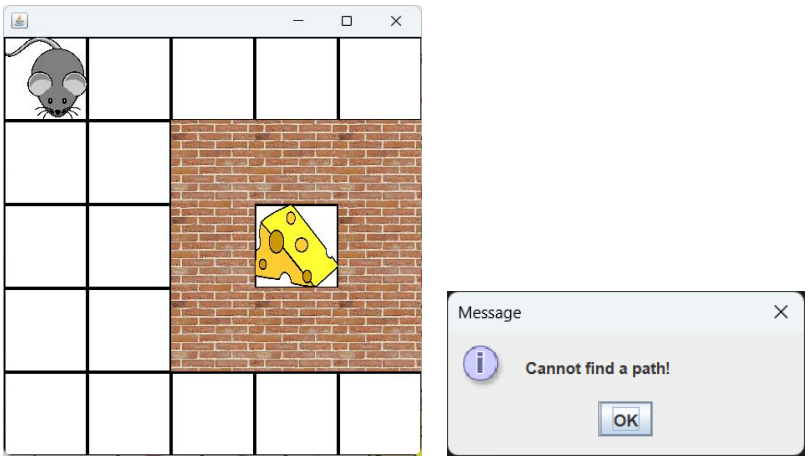
This operation should use recursion to find a suitable path. You can calculate the shortest distances to each cell using recursion and then construct the best path iteratively. There are different approaches to this problem; you can use any of them. One easy-to-implement approach is to keep track of the best direction to go into a specific cell.

For example, the recursion starts from the starting cell and visits each neighbor of the current cell. We can keep the direction we approach a cell (north, south, west, or east) and the number of cells we visit (the number of moves we need to make from the starting cell) to reach that position. When we visit a particular cell, we should update the best distance and the best direction to approach that cell only if we discover a shorter distance. The first time we visit a cell, since it has not been visited before, we can set its best distance as the current distance. However, if the cell is visited before, we should check whether the current distance is better than the best. We return if we visit a previously visited cell with a shorter distance or encounter a wall or the ending cell. We should also not recurse out of bounds.

When recursion finishes, we would have a direction and distance for each cell, as in the following sample image (you do not need to display such an image with arrows and distances in your implementation; it is only for the sake of the example). Using an iterative approach, we can start from the ending cell and follow the directions to reach the starting cell, which gives us one of the shortest paths to the cheese. Note that there are multiple shortest paths in this case, and the algorithm may find a different one based on the order in which you visit the neighbors of a cell. The directions will also change based on the order in which you visit the neighbors.



In the example above, we follow the directions from the ending cell to construct the path indicated with the purple line. Note that any path that follows the distances in the reverse order is a valid shortest path. Even if we do not keep the directions, we can utilize the numbers indicating the distances to construct such a path.



If the algorithm cannot find a valid path, you should display an error message using the JOptionPane.showMessageDialog method after the user clicks the “Find Path” button. The “Reset” button should clear all the walls and the displayed path and position the mouse and the cheese to their default locations.

Preliminary Submission: You will submit an early version of your solution before the final submission. This version should at least include the following:

- The GUI of the program should be complete.
- We should be able to add walls by clicking with the mouse on the maze.

You will have time to complete your solution after you submit your preliminary solution. You can consult the TAs and tutors during the lab. Do not forget to make your final submission at the end. Even if you finish the assignment in the preliminary submission, you should submit for the final submission on Moodle.

Not completing the preliminary submission on time results in 50% reduction of this assignment's final grade.