EECS 1720
LAB 3 :: Building Classes (Fields, Constructors and Methods)

Prerequisite - please ensure that you have setup your EECS account and can log into the lab machines prior to starting this lab! Ensure you have completed/understand Lab 1-2.

Lab Resources:

Java API: https://docs.oracle.com/javase/8/docs/api/
java.awt.Graphics2D (see Java API link)
java.awt package (see Java API link)
java.awt.geom (see Java API link)

ImagePackage API (RasterImage)
http://www.eecs.yorku.ca/course_archive/2022-23/W/1720/labs/lab3/doc

Lab Files: http://www.eecs.yorku.ca/course_archive/2022-23/W/1720/labs/lab3/lab3.zip

STEP 1: Importing an Archived Project

In this step, you will import an archived project into your Eclipse workspace, (the files for each lab exercise are embedded). Each question refers to a separate java file. The instructions for what is required for each question is included in the document below (STEP 2 onward). It is a good idea to create a separate workspace (e.g. "EECS1720") for this course.

a. You can download the lab project file from the link above (see Resources)
This file is a *zip file*, also known as an *archive file*. Click on the link below to open the URL in your browser. In the dialog that opens, choose **Save**, not **Open**. This file is typically saved into your *home* or *downloads* folder.

b. Open Eclipse and import this archive
IMPORTANT: this process is the same as the process you will use in lab tests, so please make sure you follow it, so that you do not have submission difficulties during the lab tests.

DO NOT DOUBLE CLICK ON THE FILE TO OPEN. By importing this project, you will add it into your *EECS1720* workspace. Do this by:

i. Open Eclipse (**Applications → Programming → Eclipse**) & set/choose your workspace
ii. Close "Welcome" tab, and navigate to **File → Import → General → Existing Projects into Workspace.** Hit "next"
iii. Choose the archive file (zip file you downloaded in (a)). After selecting, hit **Finish**, and the file will open up as a project in your Project Explorer.
iv. We are now ready to proceed with the Lab Exercises.

STEP 2: Lab Exercises

The exercises in this lab are pitched at creating three versions of a single class (Ghost), that will support the instantiation and graphical representation of a ghost from the classic "Pacman" arcade game from 1980:

https://www.youtube.com/watch?v=dScq4P5gn4A&t=493s

Firstly, we will create a (dynamic) class that can instantiate "**Ghost**" objects (Exercises 01-03), representing a basic Ghost character in its normal visual state (i.e. chase/scatter), where they are generally moving around looking for Pacman. The 4 ghosts in a typical game of Pacman are described in more detail in the following links:

https://en.wikipedia.org/wiki/Ghosts_(Pac-Man)
https://www.youtube.com/watch?v=ataGotQ7ir8
*** the 2nd link shows how behaviour states look (timestamp 0:00 - 1:50 only) – don't worry about any of the other details about algorithms for movement etc.*

Secondly, we will add functionality to draw a Ghost (based on its current internal state). For this the RasterImage class (discussed in lectures) will be used, and specifically, its Graphics2D reference (which gives access to fill & draw methods – see Lecture 7/8 slides). The look of the ghost will depend on whether it is normal, frightened, or eaten, and what direction it is currently facing/moving.

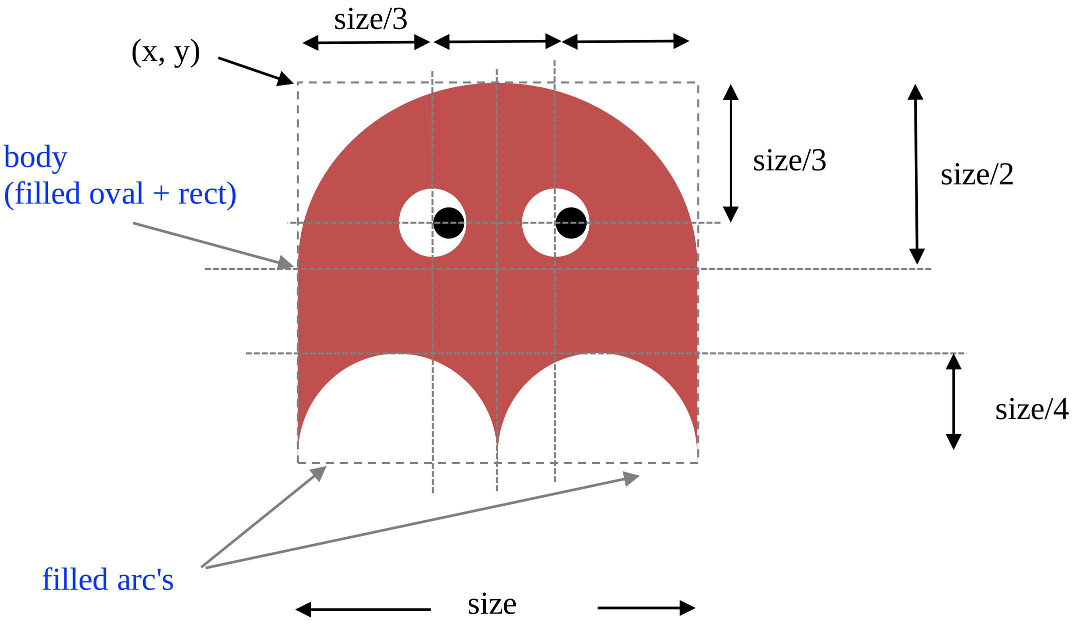
Finally, we will use the main method of a client class (Game), in which some useful constants are already provided, in order to instantiate and display the 4 ghost characters in each of the 4 corners of an app window (again using RasterImage). We will not build a working game in this lab, only the graphical elements (as classes), so that they support the creation of these objects, and how they are drawn. The ghosts will be instantiated and stored in an array, and that array used to draw and re-draw the 4 ghosts, after user input is provided to specify and move one of the 4 ghosts in the scene.

We will talk about aggregation as a concept more specifically in later lectures. For now, note that many classes aggregate other reference types (instances of other dynamic classes, or references to other utility classes) together (as class fields). For example, `RasterImage` "aggregates" a `Graphics2D` object, your Ghost classes will aggregate Color objects, and your Game class will aggregate several Ghost objects (these constitute "relationships" or "connections" that exist between these classes).

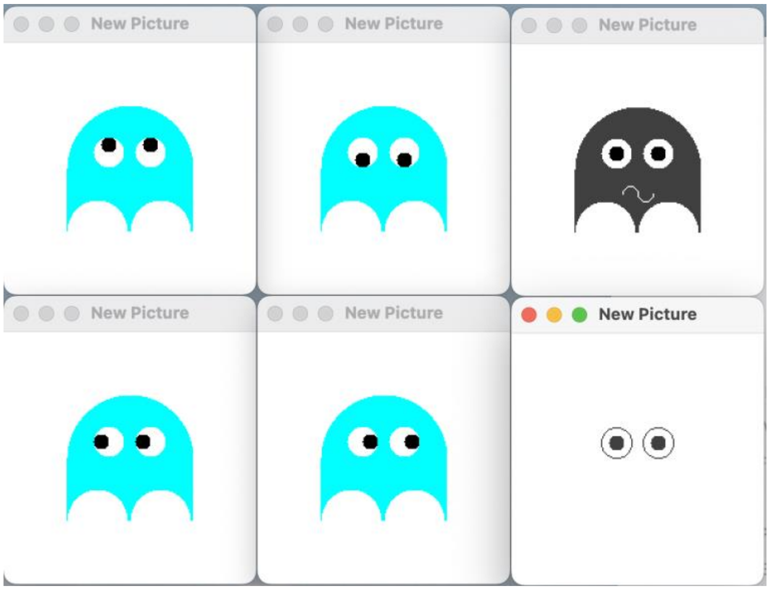
Question 01 (working with constructors – Ghost1.java):

Goal: To create a class `Ghost1`, that stores the properties (position, size, colour, frightened & eaten states, and direction) of a Ghost character that might be used in a Pacman game.

Typically, a generic ghost (in its normal state), would look something like the following (constructed from a combination of layered filled ovals and rectangles):



Where a ghost is enclosed withing a square of (size * size), and positioned at the coordinate (x,y), which represents the top left corner in the app window coordinate system (i.e. image coordinates, where 0,0 is the top left of the RasterImage window). The eyes each have a radius (eyeRad) for the whites of the eyes, and a radius (pupilRad) for the blacks of the eyes. The ghost has a direction (which reflects which direction it is currently moving – which influences the position of the pupils in the eyes, and finally, two booleans to represent the state of the ghost (to indicate if it is "frightened" or "eaten") – which also influence its visual appearance.



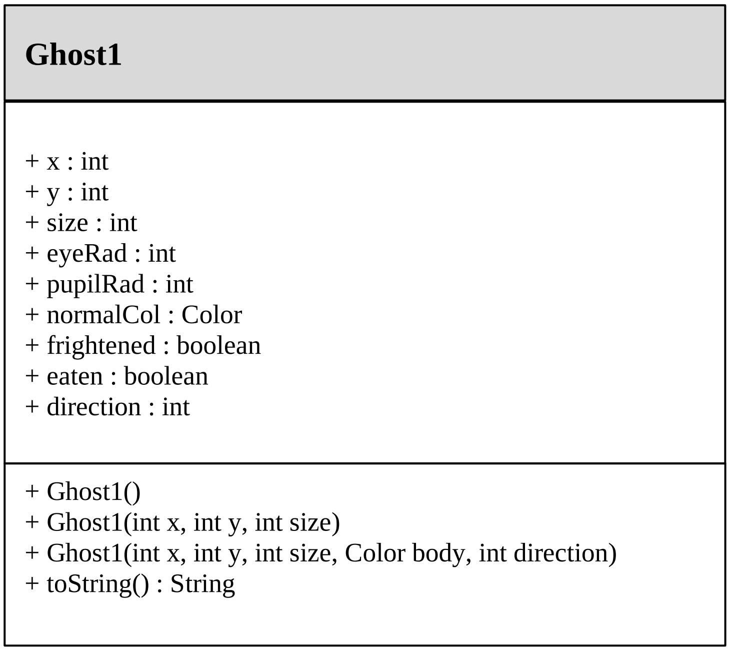
directions indicated by pupils (left and middle),
frightened (top right), and eaten (bottom right)

The `Ghost1` class should support 3 constructors, that each create and store the basic properties of a ghost (as provided in the UML diagram below):

1. A default constructor (no arguments)
   a. Default position (x,y) = (0,0)
   b. Default size is 100
   c. Default eyeRad = 1/8th the size
   d. Default pupilRad = 1/2 the eyeRad
   e. Default normalCol = CYAN (use constant from java.awt.Color)
   f. Default frightened = false
   g. Default eaten = false
   h. Default direction = EAST (i.e. 0 degrees – see/use constants in Game class)

2. Two custom constructors (according to the UML diagram below)
   a. Where values not provided as arguments are set to defaults as per (1) above

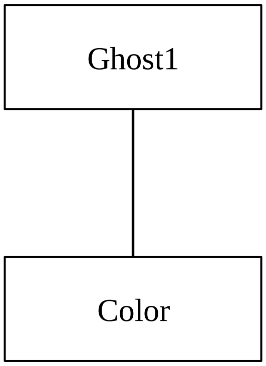
***** be sure to use the "this" keyword in your constructors (see Lecture7/8)***



Ghost1

+ x : int
+ y : int
+ size : int
+ eyeRad : int
+ pupilRad : int
+ normalCol : Color
+ frightened : boolean
+ eaten : boolean
+ direction : int

+ Ghost1()
+ Ghost1(int x, int y, int size)
+ Ghost1(int x, int y, int size, Color body, int direction)
+ toString() : String



UML representation of aggregation
"Ghost1 aggregates a Color object"

Use your main method (in Ghost1.java) as a client to test the above manually by:

- constructing a Ghost1 object called "scarey1" using the default constructor
  - print out its state using the `toString()` method provided in the shell code
- constructing a Ghost1 object called "scarey2" using the default constructor:
  - directly increase its y position by 100,
  - change its body color to red,
  - print out its state using `toString()`
- constructing a Ghost1 object called "scarey3" & "scarey4" using the custom constructors:

- scarey3 should have be instantiated with (x,y)=(300,400) and size=200 (with the remaining properties according to defaults)
- modify scarey3 after creation, to give it a light gray body colour (see Color class)
- print out scarey3 using `toString()`
- scarey4 should have be instantiated with (x,y)=(580,340), and size=180, have a magenta body, and direction SOUTH (see Game class for direction constants)
- print out scarey4 using `toString()`

  *** you may also run the PartialTester.java file to run some partial junit tests to check the functionality of methods in this lab*

After running your main, you should have the following output:

```

Ghost @ (0, 0):
      [ size = 100 * 100 ]
      [ colour = java.awt.Color[r=0,g=255,b=255] ]
      [ state = normal ]

Ghost @ (0, 100):
      [ size = 100 * 100 ]
      [ colour = java.awt.Color[r=255,g=0,b=0] ]
      [ state = normal ]

Ghost @ (300, 400):
      [ size = 200 * 200 ]
      [ colour = java.awt.Color[r=192,g=192,b=192] ]
      [ state = normal ]

Ghost @ (580, 340):
      [ size = 180 * 180 ]
      [ colour = java.awt.Color[r=255,g=0,b=255] ]
      [ state = normal ]
```

** Note, if scarey4 was set to have frightened=true, eaten=false its output would be:

```
Ghost @ (580, 340):
      [ size = 180 * 180 ]
      [ colour = java.awt.Color[r=255,g=0,b=255] ]
      [ state = frightened ]
```

If scarey4 was set to have frightened=true and eaten=true, its output would be:

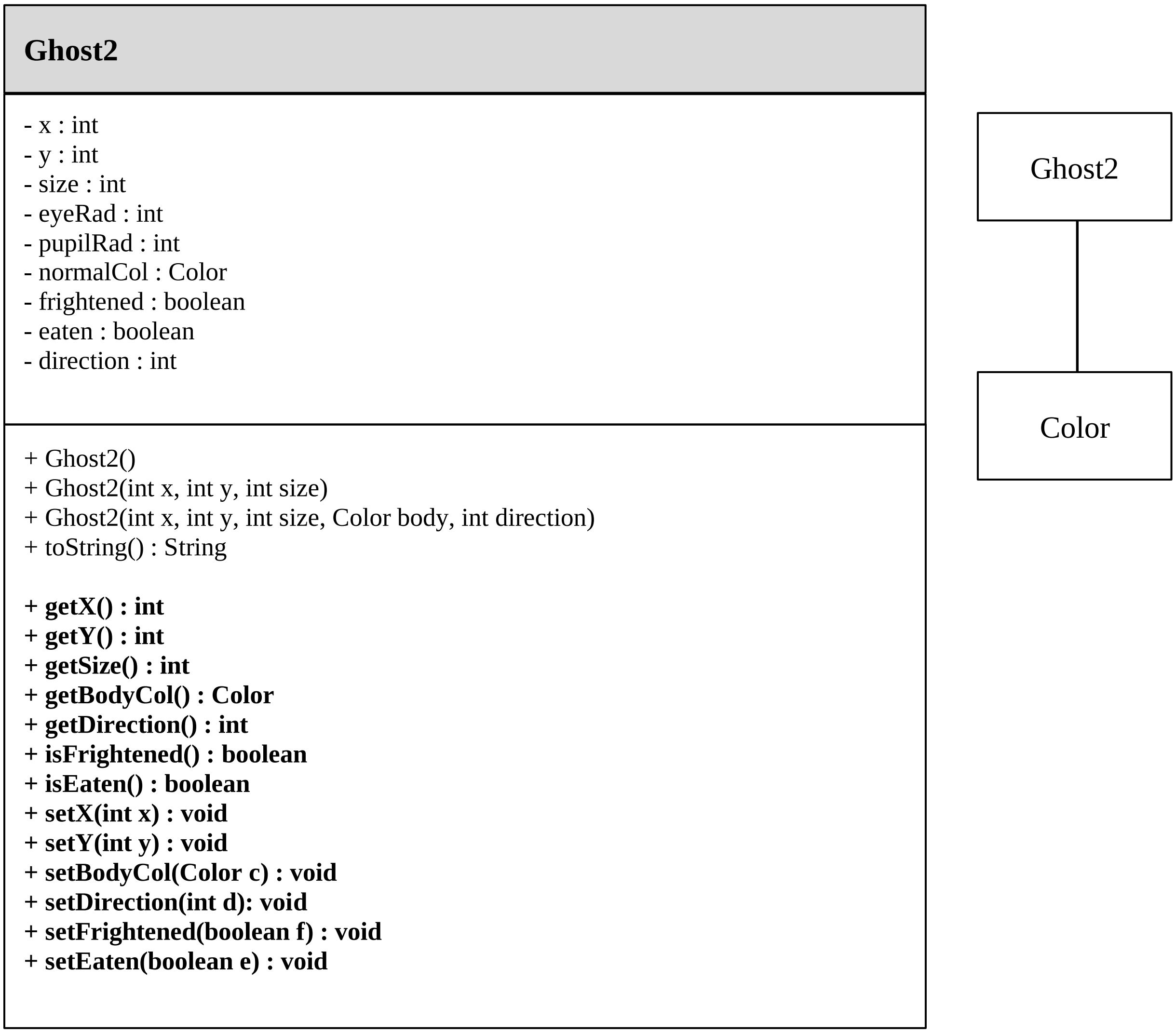
```
Ghost @ (580, 340):
      [ size = 180 * 180 ]
      [ colour = java.awt.Color[r=255,g=0,b=255] ]
      [ state = eaten ]
```

For all other cases of eaten (if frightened=false), the state would remain “normal”

Question 02 (working with access, accessors and mutation – Ghost2.java):

In this task, you will modify your Ghost1 class from part 1 to create the class Ghost2 (see UML below). To achieve this, you will need to:

1. copy over your code from inside each of the constructors from part 1 (Ghost1.java) – into the respective constructors for part1 - note they will be named Ghost2(..) now, not Ghost1(..).
2. modify the "access" properties for class variables as per the UML diagram below
3. create the relevant accessors and mutators for the Ghost2 class (in bold)
    a. each accessor should **get** (return) the value of the associated field
        i. note the two isXX() methods are accessors that return the boolean fields
    b. each mutator should use its argument to **set** the value of the associated field



Ghost2

- x : int
- y : int
- size : int
- eyeRad : int
- pupilRad : int
- normalCol : Color
- frightened : boolean
- eaten : boolean
- direction : int

+ Ghost2()
+ Ghost2(int x, int y, int size)
+ Ghost2(int x, int y, int size, Color body, int direction)
+ toString() : String

+ getX() : int
+ getY() : int
+ getSize() : int
+ getBodyCol() : Color
+ getDirection() : int
+ isFrightened() : boolean
+ isEaten() : boolean
+ setX(int x) : void
+ setY(int y) : void
+ setBodyCol(Color c) : void
+ setDirection(int d): void
+ setFrightened(boolean f) : void
+ setEaten(boolean e) : void

4. Use the main method in a **client application** (Ghost2Client.java) to test the above by:
    a. constructing the same Ghost objects as you did in part 1 (as Ghost2 objects) (you will need to use the mutators for scarey2 and scarey3)

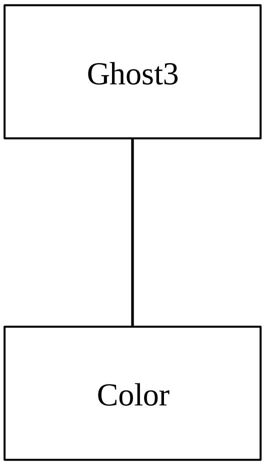
b. print out all Ghost2 objects using toString() to check them

*** you may also run the PartialTester.java file to run some partial junit tests to check the functionality of methods for this question*

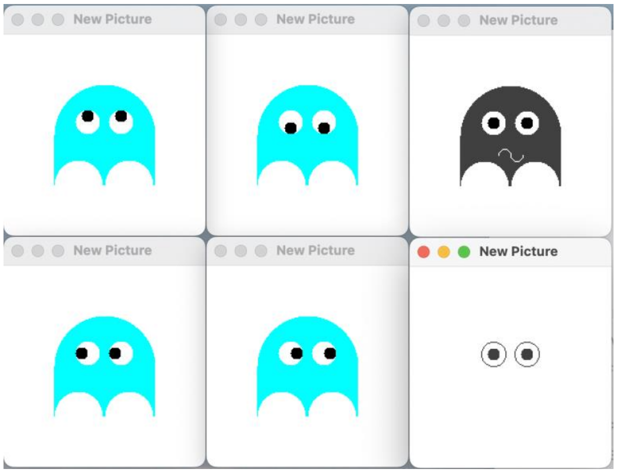
Question 03 (render & move methods): Ghost3.java

In this task, add a **render()** method to your Ghost2 class to create the Ghost3 class below. Make sure you copy over the code from part2, and rename the constructors to Ghost3(..). The render() method will likely take the most time for this lab (please do not do this at the last minute).

Ghost3
// previous fields not shown (follows previous diagram)
// previous methods not shown (follows previous diagram) **+ render(Graphics2D g) : void** **+ move(char dir, int step, RasterImage r) : boolean**



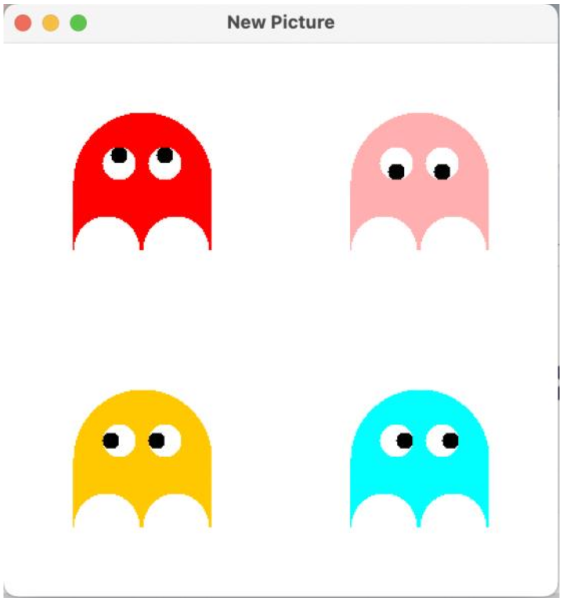
Note that to render a Ghost3 object, the size parameters and body colour must be according to the properties of that object. The proportions are roughly set out in the first figure of this document (you do not have to adhere to this exactly, but the proportions should help you figure out relative coordinates for the Ghost). Note also, that the direction property is indicated by the location of the pupil in the eye – i.e. NORTH would have the ghost looking UP (pupils at top centre), EAST would have the pupils looking to the right, WEST pupils would be looking left, and SOUTH pupils would look down.



If the ghost is frightened (but not eaten), the ghost would appear to have a dark gray body, pupils centred, and a scared mouth (which is not present in normal state). If the ghost is frightened and eaten, then only the eyes of the frightened ghost would be rendered (see figure above). You have some creative license here, the look does not have to be EXACT.

- In the main method of the Game.java file (Game class), create a `RasterImage` object with a size of 800 (h) x 800 (w) pixels.
- Include a calls to create four `Ghost3` objects (for each of the 4 ghosts in the game: i.e. "shadow", "pinky", "inky" and "clyde"). Position each in the middle of each of the 4 quadrants of the RasterImage window (e.g. "shadow" is in the top left), and ensure set their sizes to ¼ of the size of the RasterImage window. Note, if you change their size (they should scale accordingly, but still be positioned in the centre of each quadrant).
- Store these objects in a single array (i.e. has 4 elements).
- Now make calls to render() on every element in the array. It is useful to clear the screen inside your draw method before drawing, that way if you change a property and call render(), any previously drawn versions will be removed from the screen

The result should look something like this (for a `RasterImage` of size 800 x 800) – assuming the ghosts are all initially in a non-frightened, and non-eaten state. You can experiment with the frightened/eaten states by using the mutator for each to set and redraw.



<u>Optionally (if time permits)</u>, you may also add a **move** method (to mutate the position of the Ghost3 object left/right/up/down on the screen by a certain amount of pixels, where the char "dir" is either 'L','R','U', or 'D' indicating left, right, up or down respectively); and "step" represents the number of pixels to increment in the move in that direction. The method should return a boolean to indicate if the Ghost3 object's position (x,y) is on the screen (true) or off the screen (false). [Hint: you can use RasterImage class methods to test this]. Test your move method by initializing the positions of the elements as above, and then invoking the move method on

** NOTE: there are no testers for the render() and move() methods → as you are able to render them according to your own aesthetic, as long as the eyes indicate the directions for the normal state, and you have a version for the frightened and eaten states.

STEP 3: Submission (Deadline: Tuesday 7th February, 11:59pm)

You will formally submit the following java files (associated with all Questions01-03): **Ghost1.java**, **Ghost2.java**, **Ghost2Client.java**, **Ghost3.java**, **Game.java** & **PartialTester.java**

SUBMIT ALL OF THE *.java FILES (**NO ZIP FILES OR CLASS FILES!!**).

Recall: if you submit multiple times, the files previously submitted will be overwritten with the newer versions of the files. You can submit an unlimited number of times up until the deadline for this lab.

Web-submit can be found at the link: https://webapp.eecs.yorku.ca/submit/

** submission will be closed immediately after the deadline. Ensure you submit early and often up until you complete the lab, it is not suggested to wait until the last minute.