Design Document Office Dude VS. Slugs CSC 171

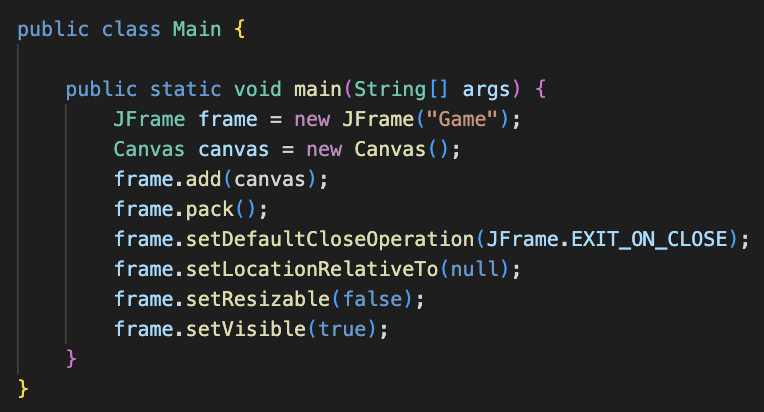
Leanna Fowler, Kebron Zewda, Lexi (Sunny) Wu, Kyle Jhong

**Class Main**

The main class contains the main method:

public static void main(String[] args)

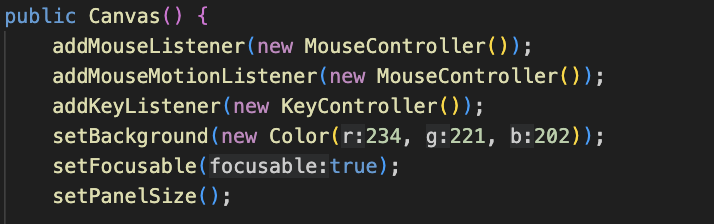
In this method, we instantiate a JFrame and Canvas (which is a class that we made) and make everything visible using graphics.

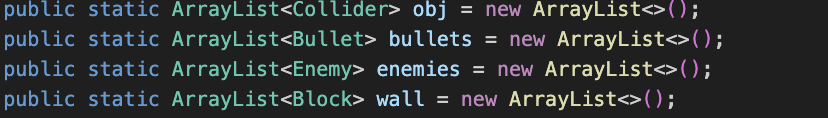


**Class Canvas**

CONSTRUCTOR

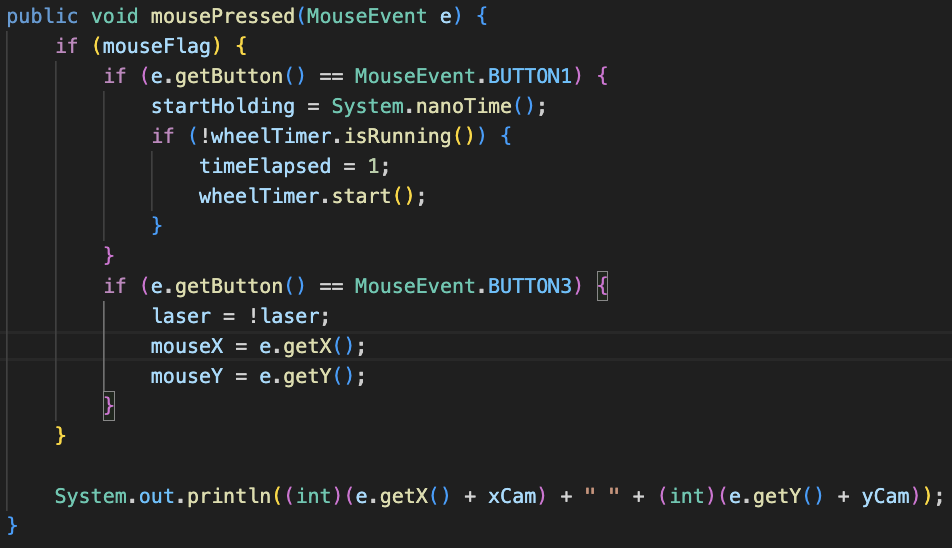
This class is a massive class that keeps track of all inputs, timers, and positions. Additionally, this class holds array lists for all Colliders, Bullets, Enemies, and Blocks in the game.

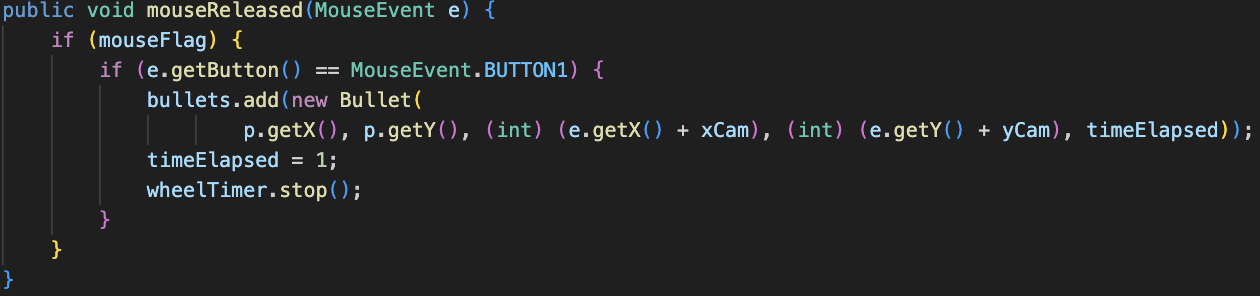




We added our MouseListener and KeyListener in this class, as well as initialized the array list for Blocks, created the player object, and set up the physics timer, charge-up timer, and enemy wave timer – these are instantiated with inner classes.

INPUTS





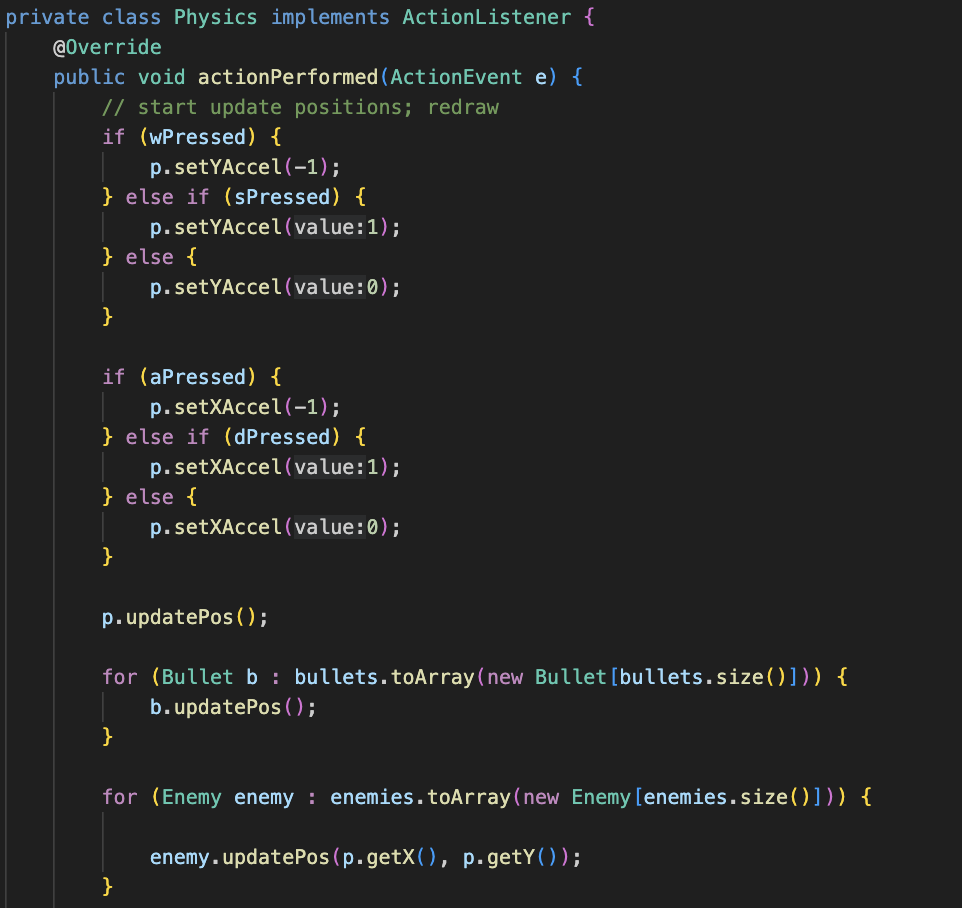
For inputs, we created an inner class called MouseController which implements MouseListener. When the mouse is pressed, we start our charge-up timer, which depending on how long the player presses down the mouse, more or less damages the bullet. Consequently, when the mouse is released, it calculates the time elapsed since the mouse was pressed to figure out how powerful to make the bullet, as stated before). In addition, it adds a new Bullet object to the array list, passing in the player’s position, the mouse position, and the time elapsed.

By using the methods mouseReleased and mousePressed, we can successfully monitor the changes that are happening when the mouse is pressed or released, making it easier for us to debug or add to our code when we wish to implement something new.

PHYSICS — MOVING OBJECTS

Simply, this inner class adds physics to our game. Naturally, when one begins to run, they don’t run at top speed immediately. Thus, to make our game more realistic, we made sure our Dude does not accelerate to full speed when switching directions by slowly increasing their speed until it hits maximum speed.

To perform this, we made booleans for each key input (W, A, S, D). Depending on the values of these booleans, the program calls the Player’s setXAccel and setYAccel function. Then, the program calls the updatePosition function of every moving object in the game (Player, Bullet, Enemy)



PHYSICS — CAMERA

Additionally, we created a camera system where the player is always in the center of the screen. There are two static variables: xCam and yCam. Each is defined by:

xCam = p.getPosX - (width of screen / 2)

yCam = p.getPosY - (height of screen / 2)

Then, we make sure the camera stays in the bounds of the

To make this happen, we coded the program so every object (Player, Bullet, Enemy) stores its position. Then for each of these objects, to determine where they should be positioned on the screen relative to the camera- they draw themselves at position - camera position using their own draw() function.



PAINTCOMPONENT

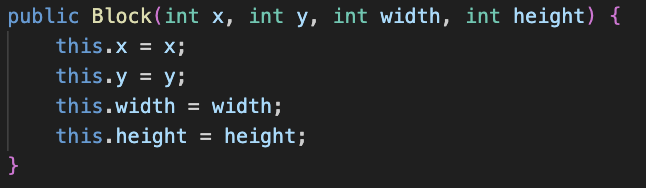
To draw everything on the screen, we used the PaintComponent method. Depending on the state of the player (death or not death), the screen draws different things. When the player is still alive, the paint component method is called whenever the player moves (which makes all the other components on the JPanel move as well). This is done by calling repaint().

SCORE

For the game to be complete, we implemented a high score mechanism. Whenever the player dies, the screen dims and on the left-hand side of the screen, the high score list of the top 5 scores is displayed in descending order. The score class (which holds together a score and which attempt the score is associated with) implements Comparable and sorts itself based on the score.

**Block Class**

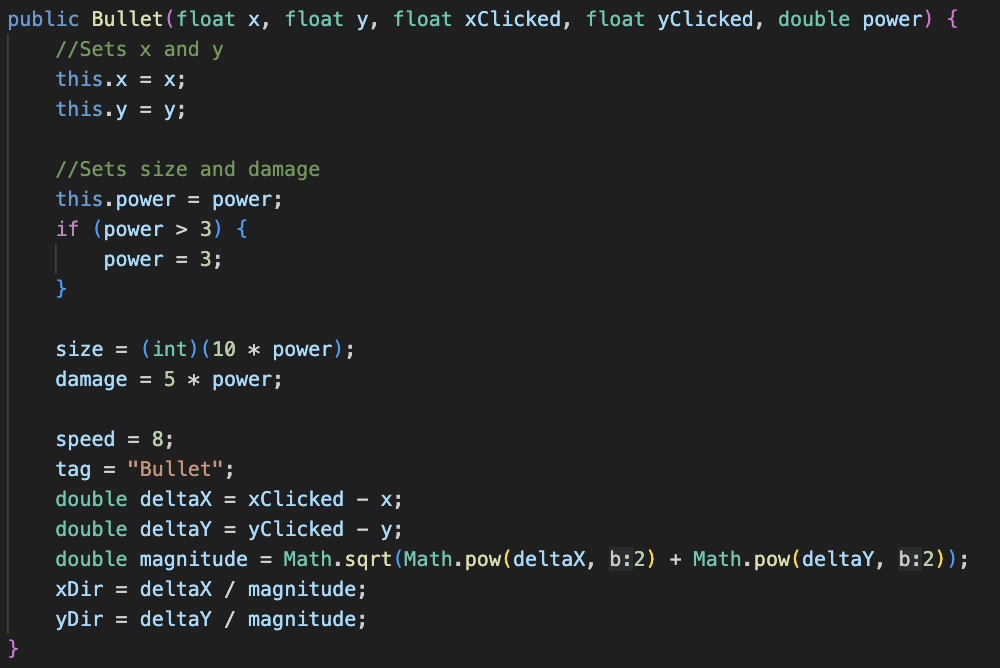
This short class is designed to make the walls of the map.



The constructor takes in 4 parameters: int x, int y, int width, and int height. There is a drawing method called draw() that draws the block using each of the parameters.

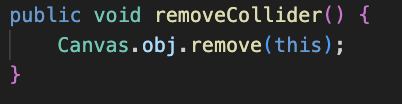
**Bullet Class**

This class is designed to create the bullets and modify them, along with the object that each bullet collides with.



The constructor takes 5 parameters: float x, float y, float xClicked, float yClicked, double power. The xClicked and yClicked are floats that represent where the mouse was clicked on the screen. The constructor uses these variables to figure out which direction the bullet should move in (storing the unit vector in variables xDir and yDir). The power represents the time elapsed between the moment the mouse is pressed and released, and depending on the power, the bullet is more powerful or vice versa.

Another addition to the bullet class is that we have to consider that the bullet will collide with another object. Due to this fact, we created a method that will simply remove the object hit from the array list obj, which is an array list of Collider (this class will be next in the document)



Lastly, this class extends the Collider class, which will be discussed next.



**Collider Class**

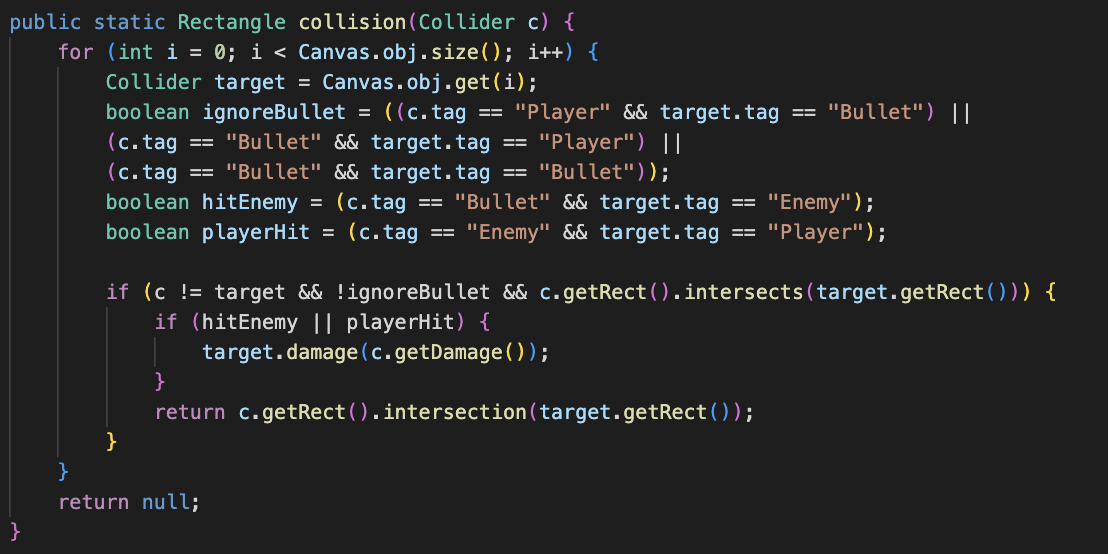
This abstract class is designed to handle all the Colliders in the program. What this means is that this class’ job is to monitor what is colliding with what.

Every object on screen in the program extends the Collider class. They inherit a String variable called “tag”, and the tag is set to the class name itself inside every child class. For instance, the player has the tag “player”, the bullet has the tag “bullet”, etc.

The method collision() takes in a Collider of a particular object, and calculates whether it overlaps with another object, returning a Rectangle object of the overlap between this collider and another. It checks the Collider that is passed in against every other Collider in the scene. It returns a Rectangle object of the overlap so that when the Player and Enemy moves into walls, they can move backwards the exact distance they overlapped (which puts them right on the edge). The method collision() also makes sure that the player doesn’t collide with its own bullets. The script checks to see if the Collider that is passed in and the current target collider it is comparing have the “Player” and “Bullet” tags, and if they collide, it returns null instead of a Rectangle of their overlap.

This class also holds two empty functions, void damage(double damage) and double getDamage(). Certain subclasses override these functions: for example, the Player overrides the damage(double damage) function, which subtracts the passed in value from its health. The bullet overrides the getDamage() function, which returns how much damage it should inflict.

Then, in collision(), we then use if statements to call these functions, but only if it is a Bullet hitting an Enemy or an Enemy hitting a Player (which we can find using the tags). If the player is hit, then the player will be damaged, and if the enemy is hit, the enemy will be damaged, as demonstrated in the following code.

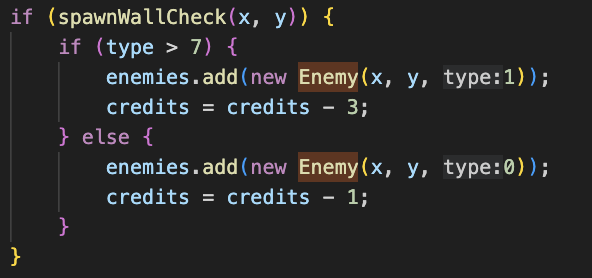


**Enemy Class**

This class is designed to manage all the enemies in the game.

The constructor takes in 3 parameters: float x, float y, and int type. The variable type has two possible values - 0 or 1. If it is type 0, the red slug will spawn (with its corresponding stats); if it is type 1, the beige slug will spawn (with its corresponding stats)

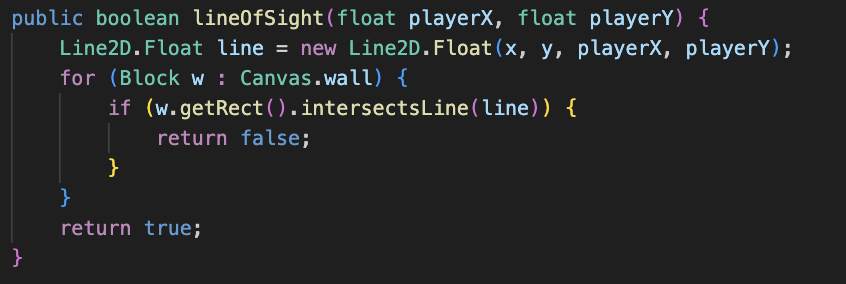
→ this is done in the MAIN class:



*Extra: The credit system*

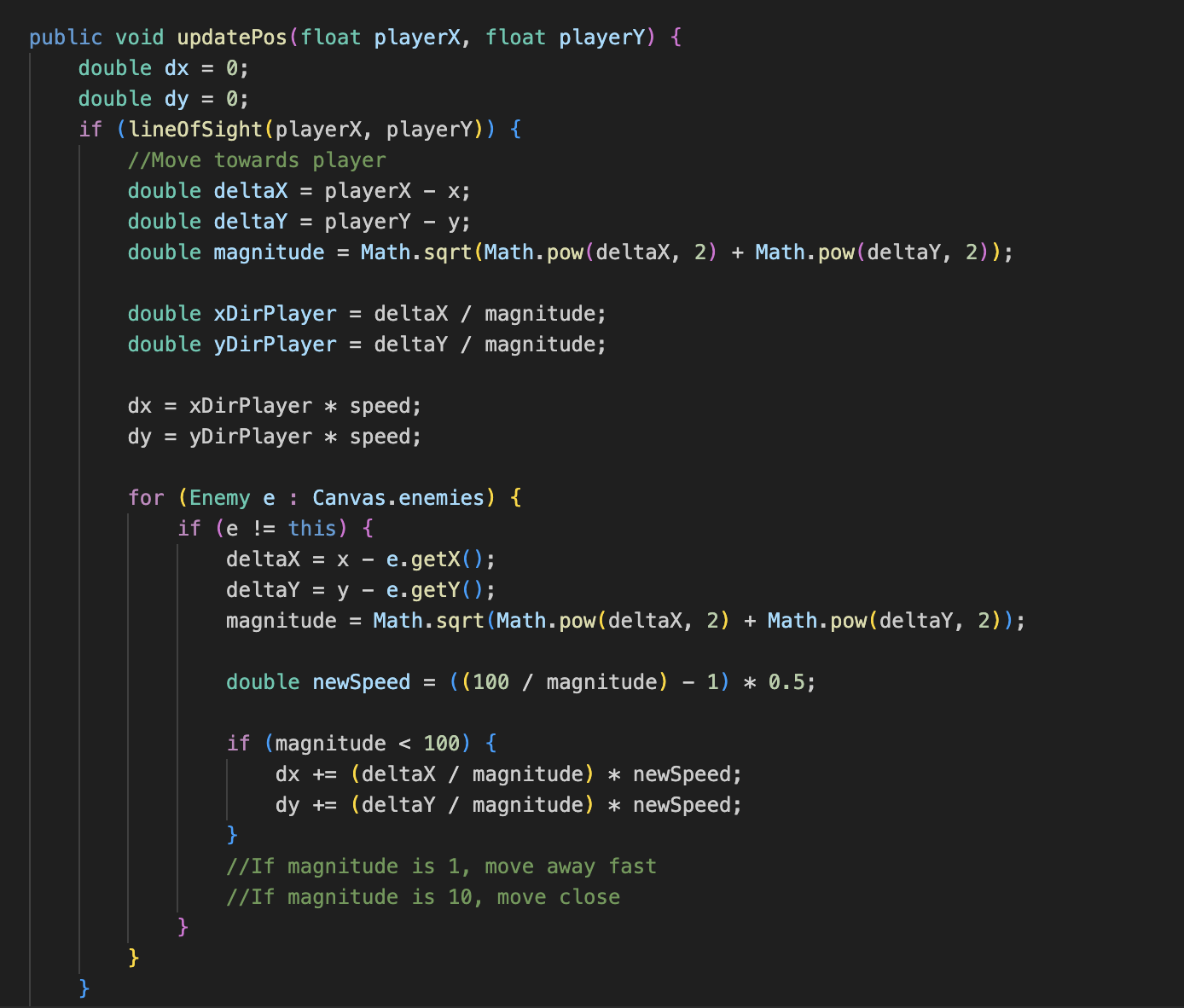
*This system ensures that the correct slug is spawned, more specifically, randomly spawned. To truly randomize the spawning of the slugs, we randomly generated numbers with a bound of 10. If this number exceeds 7, a beige slug will spawn, and vice versa; the probability of the beige slug spawning is less than that of the red slug.*

To enhance the game, we implemented simple enemy AI: line of sight.

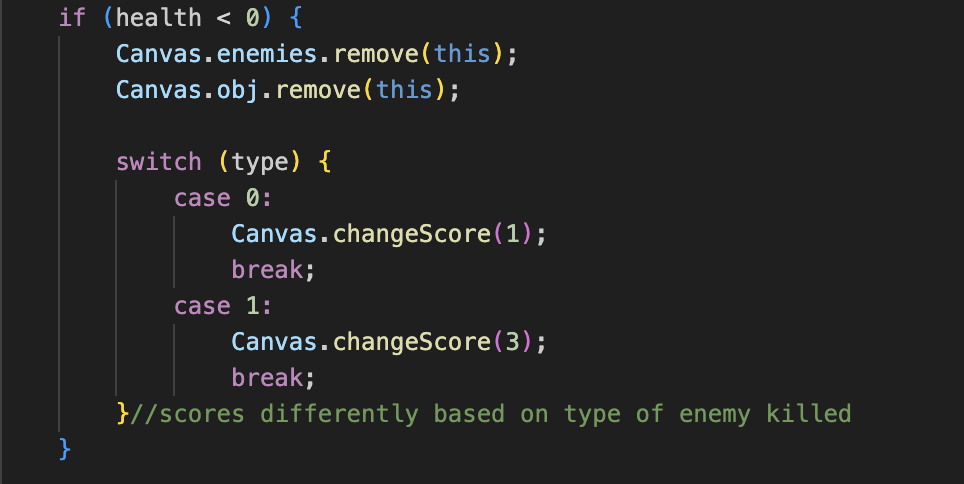


In short, this method allows the slugs to follow the player whenever it “sees” the player. This is done by using Line2D. Whenever the line intersects with a wall, this method, called lineOfSight will return false, resulting in the slug not moving. Otherwise, the method will return true, and the slug will move towards the player.

updatePos() is the main method in the enemy class. It is called with an enhanced for loop from the Physics inner class in Main. If the lineOfSight() function returns true, it moves the enemy towards the Player. The updating of the position and collision detection is done similar to the player class (which will be detailed later), but it has a certain addition that makes it so that the enemies space out from each other if they are too close to each other. It iterates through all the enemies in the arraylist, and if too close together, moves away from the other enemies.



Enemy death: A private double called health holds the enemies health. As the enemy is hit with bullets, the health goes down. Eventually, once the health reaches less than 0, the enemy dies and the score is updated for the player and the enemy is removed from the canvas.



Lastly, this class extends the Collider class, which will be discussed next.

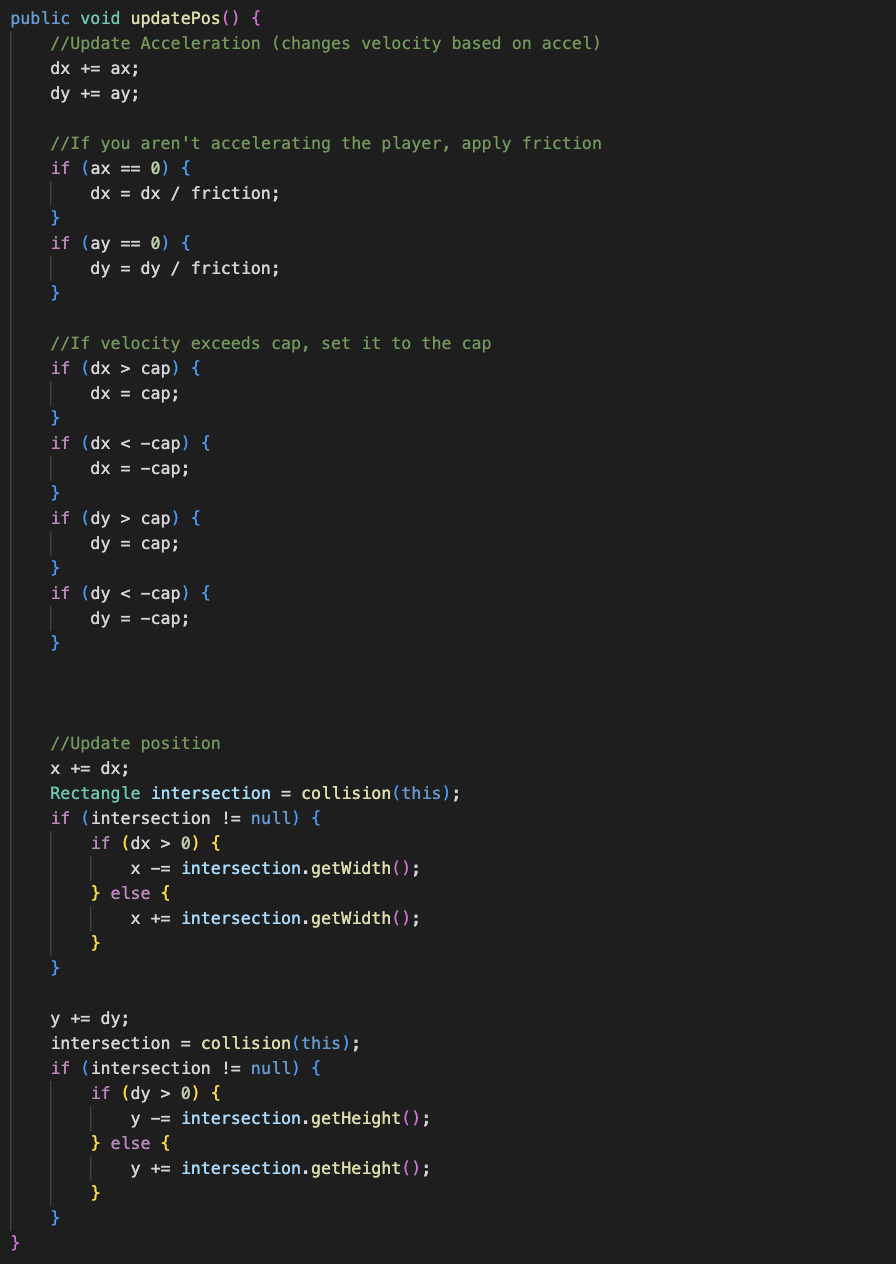


**Player Class**

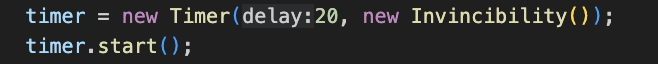
The player class is designed to manage the player in the game.

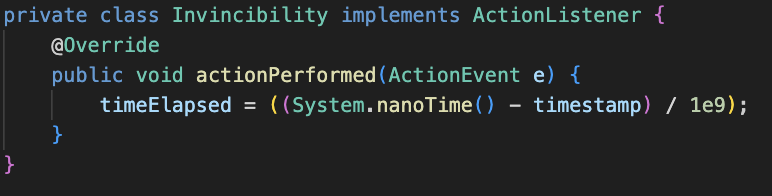
The constructor takes in 2 parameters: float x and float y. Both these variables represent the current location of the player.

There are two functions used to set the x and y acceleration of the player, setXAccel(float value) and setYAccel(float value). The Physics inner class in the Main class updates these variables to make the player move. These get taken into consideration in the updatePos() function, which gets called from Physics in Main as well. This function takes the acceleration and updates the velocity every frame, and then caps the velocity at a certain value (by checking if it exceeds a certain value and setting it back to that value). This way, the player will accelerate from rest until it hits a capped velocity. Then, it updates the position based on the velocity. Then, it checks if it collided with anything (using the collision function from the Collider class), and if so, it pushes the player backwards by the amount it clipped into the other object.

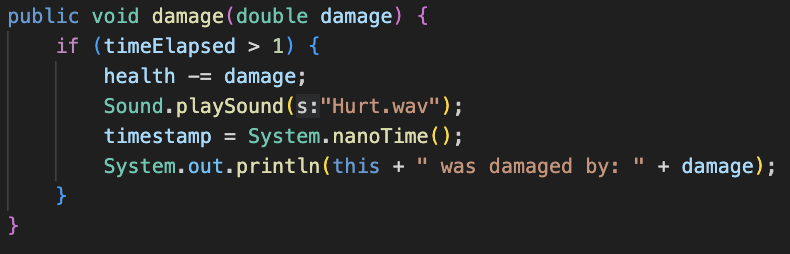


Not only does this class manage the position, health, collision, and damage, but it also incorporates a damage cooldown, called Invincibility that implements ActionListener.



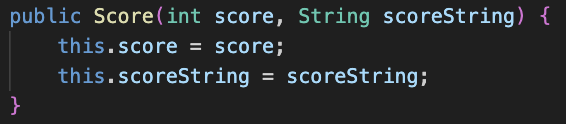


Rather than the health always depleting whenever the enemy is colliding with the player, the timeElasped variable ensures that the player will get damaged every second. This function makes the game more enjoyable, potentially helping the player more *invincible*.



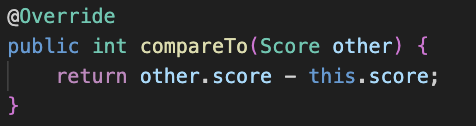
**Score Class**

This short class allows us to create a list of scores and order the scores in descending order using the interface comparable.



The constructor takes in 2 parameters: int score, String scoreString. The score represents the score that the player cumulates over the game, and the scoreString represents the following phrase: "Play through " + numberTimesPlayed + ": " + score + " points", where numberTimesPlayed is and integer that keeps track of the current player through count.

To order the scores so that it is in descending order, we used Comparable. The interface requires us to use the method compareTo(), where we pass through Score other, representing another score in the highScoresAList.



If the this.score is larger than other.score, then this method will return a negative number, resulting in this.score being placed before other.score, and vice versa.

**Sound Class**

This last class is designed to take care of the sound effects in the game.

It’s rather simple, the class only has one static method called playSound() that takes in a String s (the file name). Using try-catch statements, it tries to search for a file in the directory with the given string. Then, we use the AudioInputStream and Clip classes from javax.sound.sampled to play the sound.

**How each class relates to each other**

Refer to the diagram below:

