HW4 CODE REVIEW
Austin Yu and Annie Boltwood

[CODE REVIEW - REFACTORING CODE WRITTEN BY AUSTIN]
Author of this section: Annie
[1] Random values are hard-coded into the method **resetToRandomCoordinate()**, in the **Reward** class. This was corrected by declaring these random values as constants in the Enum class Constants.

```
public void resetToRandomCoordinate(){
    Random rand = new Random();
    int randomX = rand.nextInt( bound: 23) * 50;
    int randomY = rand.nextInt( bound: 23) * 50;
    this.x = randomX;
    this.y = randomY;

}
```

```
public static final int RANDOM_VALUE = 23;
```

```
    Random rand = new Random();
    int randomX = rand.nextInt(Constants.RANDOM_VALUE) * 50;
    int randomY = rand.nextInt(Constants.RANDOM_VALUE) * 50;
```

After correction, the random value is given a much larger scope and is no longer hard-coded into the resetToRandomCoordinate() method.
[2] In the class **EntityFactory**, there are various repeated values inside of method **createPunishment()** used to scale the wall size values and the random bounds.

```
for (int i = 0; i < 4; i++) {
    while (map.getEntityAt(randomX, randomY) != null) {
        randomX = 2*Constants.WALL_SIZE + rand.nextInt( bound: 20) * 50;
        randomY = 3*Constants.WALL_SIZE + rand.nextInt( bound: 11) * 50;
    }
```

```
int wallScale1 = 2;
int wallScale2 = 3;
int randomBound1 = 20;
int randomBound2 = 11;
```

So in order to refactor this, I created variables in the method that were declared at the top rather than inside of individual lines. After this replacement the function is a lot neater since the values are given names, which will make it easier for other developers to identify what they do.
[3] After refactoring **createPunishment()** in the previous step, I noticed that the creation of random coordinates occurred in other methods in the class **EntityFactory** as well. Methods **createReward()**, **createMovingEnemy()**, and **createPunishment()** all generated random coordinate values, meaning that rather than repeating code, it would be more efficient to create a method that could be called to generate these random coordinates.

```
Random rand = new Random();
int randomX = wallScale1*Constants.WALL_SIZE + rand.nextInt( bound: 20) * 50;
int randomY = wallScale2*Constants.WALL_SIZE + rand.nextInt( bound: 11) * 50;
```

I built the function **getRandomPosition()** which is called in the 3 classes mentioned above.

```
private int getRandomPosition(Random rand, int base, int range) {
    return base + rand.nextInt(range) * 50;
}

int randomX = getRandomPosition(rand,  base: wallScale1 * Constants.WALL_SIZE, randomBound1);
int randomY = getRandomPosition(rand,  base: wallScale2 * Constants.WALL_SIZE, randomBound2);
```

4. Finally, in the **EntityFactory** class, there were three separate loops that generated the barriers of the game. However, these loops could be combined into one loop while still maintaining the same functionality. With there being three of these loops, there was a lot of repeated code that could be shortened to reduce complexity and readability of the code. Shown below is the refactored version of all three loops reduced down into one loop.

```
//barrier generation algorithm here
// Filling the maze with randomly placed barriers
// Barrier generation algorithm
for (int i = 0; i < Constants.SCREEN_WIDTH; i += 2 * Constants.WALL_SIZE) {
    for (int j = Constants.WALL_SIZE * 2; j < Constants.SCREEN_HEIGHT; j += 2 * Constants.WALL_SIZE) {
        if (map.getEntityAt( x: i + Constants.WALL_SIZE, j) == null && rand.nextInt( bound: 4) == 1) {
            barriers.add(new Barrier(Constants.WALL_SIZE,  x: i + Constants.WALL_SIZE, j, Constants.WALL_SIZE, Constants.WALL_SIZE));
            map.setEntityAt( x: i + Constants.WALL_SIZE, j, barriers.get(barriers.size() - 1));
        }

        if (i < Constants.SCREEN_WIDTH - 2 * Constants.WALL_SIZE && map.getEntityAt(i,  y: j + Constants.WALL_SIZE) == null && rand.nextInt( bound: 4) == 1) {
            barriers.add(new Barrier(Constants.WALL_SIZE, i,  y: j + Constants.WALL_SIZE, Constants.WALL_SIZE, Constants.WALL_SIZE));
            map.setEntityAt(i,  y: j + Constants.WALL_SIZE, barriers.get(barriers.size() - 1));
        }

        if (map.getEntityAt(i, j) == null) {
            barriers.add(new Barrier(Constants.WALL_SIZE, i, j, Constants.WALL_SIZE, Constants.WALL_SIZE));
            map.setEntityAt(i, j, barriers.get(barriers.size() - 1));
        }
    }
}
```

[CODE REVIEW - REFACTORING CODE WRITTEN BY ANNIE]
Author of this section: Austin

1.

```
for (int i = 0; i < punishments.size(); i++) {
    boolean isCollided = player.isPunishmentContact(punishments.get(i));
    if(isCollided == true){
        punishments.get(i).manageDamageDealingTimeout(clock.getTime());
        if(player.getScore() <= 0) {
            currentState = Constants.GAME_STATE.GAMEOVER;
            player.resetUniqueColours();
            spaceStation.resetSpaceStation();
        }
    }
}
```

2. The following code in the Game class has a lot of repetitive logic and repeating conditions that could be simplified. Further it uses repeated code for resetting the game

```
public void handleKeyPresses(int keyCode){
    if (keyCode == KeyEvent.VK_LEFT & currentState == Constants.GAME_STATE.PLAY) {
        player.moveLeft();
    } else if (keyCode == KeyEvent.VK_A & currentState == Constants.GAME_STATE.PLAY) {
        player.moveLeft();
    } else if (keyCode == KeyEvent.VK_RIGHT & currentState == Constants.GAME_STATE.PLAY) {
        player.moveRight();
    } else if (keyCode == KeyEvent.VK_D & currentState == Constants.GAME_STATE.PLAY) {
        player.moveRight();
    } else if (keyCode == KeyEvent.VK_UP & currentState == Constants.GAME_STATE.PLAY) {
        player.moveUp();
    } else if (keyCode == KeyEvent.VK_W & currentState == Constants.GAME_STATE.PLAY) {
        player.moveUp();
    } else if (keyCode == KeyEvent.VK_DOWN & currentState == Constants.GAME_STATE.PLAY) {
        player.moveDown();
    } else if (keyCode == KeyEvent.VK_S & currentState == Constants.GAME_STATE.PLAY) {
        player.moveDown();
    }
    else if (keyCode == KeyEvent.VK_SPACE & currentState == Constants.GAME_STATE.START_SCRE
        currentState = Constants.GAME_STATE.PLAY;
    }
    else if (keyCode == KeyEvent.VK_SPACE & currentState == Constants.GAME_STATE.GAMEOVER)
        //reset all vars
        //start off start_screen
        currentState = Constants.GAME_STATE.START_SCREEN;
        gamePanel.setCurrentState(currentState);

        player.setX(350);
        player.setY(350);
```

To simplify the code I refactored the code to use switch statements for handling different key events based on the game state. I also created a reset game method to reduce duplication and increase code maintainability. Also, conditions are organized based on the game states, making the code more readable and easier to understand.

3. Unused Comment in Regular Reward Class

```
/image = ImageIO.read(getClass().getResource("/images/character.PNG"));
ufferedImage originalImage = ImageIO.read(getClass().getResource( name: "/images/"+fileName));
```

Unnecessary comments in Main Character Class

```
/*public void printKeysStatus(){
    Enumeration<Constants.KEY_COLOR> keysEnumeration = keys_tracker.keys();

    while (keysEnumeration.hasMoreElements()) {
        Constants.KEY_COLOR key = keysEnumeration.nextElement();
        System.out.println(key + ": " + keys_tracker.get(key));
    }
}*/
```

```
/*if(xCoord < Constants.SCREEN_WIDTH - 80 & xCoord > 0 & yCoord < Constants.SCREEN_HEIGH
    return false;
}
return true;*/

**
```

I removed unnecessary comments to improve code readability

4. Variable naming could be clearer and comments are outdated with current code.I renamed the variables to be more readable and revised to comment to reflect the current code

```
public Boolean isNearBarrier(int xcoord, int ycoord) {
    int xepsilon = 35;
    int yepsilon = 70;
    for(Barrier b: barriers) {
        //the +50,+25,etc are hardcoded in values to make collision detection between player
        // barrier more smooth
        if (b.getX() < xcoord + xepsilon + 0 & b.getX() > xcoord - xepsilon &
                b.getY() < ycoord + yepsilon -30 & b.getY() > ycoord - yepsilon +30) {
            return true;

        }
    }
    return false;
```

```
lic Boolean isNearBarrier(int xCoord, int yCoord) {
 int xEpsilon = 35;
 int yEpsilon = 70;

 for(Barrier b: barriers) {
     //the -30,+30 are hardcoded in values to make collision detection between player and
     // barrier more smooth
     if (b.getX() < xCoord + xEpsilon & b.getX() > xCoord - xEpsilon &
             b.getY() < yCoord + yEpsilon -30 & b.getY() > yCoord - yEpsilon +30) {
         return true;

     }
 }
 return false;
```