REPORT OF THIS IS THE ONLY LEVEL

LINK-> https://youtu.be/TilI5jLzL_I

INTRODUCTION

This Is the Only Level is a game which player tries to reach the exit pipe by overcoming obstacles without touching the spikes as it respawns the start pipe when player character collides to spikes. The player uses keyboard inputs to move player character which is an elephant in this game by the help of clues. To reach the exit pipe, the player must unlock the door by pressing the button of the door. Death counts, help, reset and restart buttons, game time are also located in the information display which is the bottom side of the game display ***(Figure-1)***. In this implementation, the game has five stages, and these stages are the followings:

Stage 1: Standard controls (arrow keys)

Stage 2: Reversed left/right movement keys.

Stage 3: Continuous auto-jumping with no upward key input.

Stage 4: Requirement to press the button five times to unlock the door.

Stage 5: Custom controls using keys F, T, and H for movement

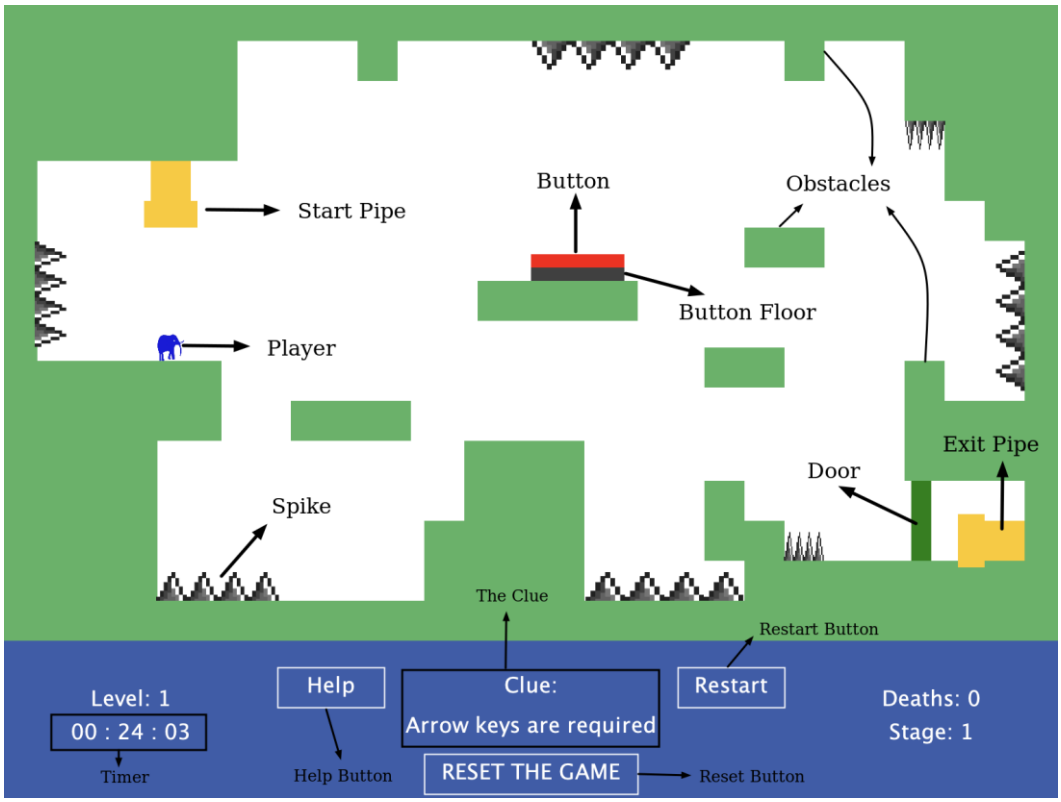


Figure-1: Game Environment

IMPLEMENTION DETAILS

Code of the game consists of 5 classes since it is coded by object-oriented-programming. These classes' general structure (except Main) has four parts, and these are data fields, constructor, setter-getter methods, and other methods.

- While explaining the classes, interactions with other classes are shown by initials of that class with brackets.

➔ Main:

This class has the main method. Stage objects are created here ***(S)***, and added an arraylist. After that, a game object is created and its play method is called for start of the game ***(G)***. All class code is shown in the ***Code-1***.

➔ Stage: *(S)*

This class holds configurations of each step such as gravity, speed, controls, clue text, help text, color, and assign color of obstacles except white as shown in ***Code-2***. Also, the class has getter methods of many data fields to be accessible from other classes as shown in ***Code-3*** *(usage will be shown in the codes of Map and Game classes).*

➔ Player: *(P)*

This class is responsible for the player character which is a blue elephant. It stores position, velocity, direction, and state (jumping, grounded, face direction) of the player and provides getter and setter methods of many data fields to be accessible and modifiable from other classes as shown in ***Code-4*** *(usage will be shown in the codes of Map and Game classes)*. Moreover, there are some methods for player's movements: jump, moveLeft, moveRight, and applyGravity ***(Code-5).*** Draw method is responsible for displaying the elephant in true direction. Respawn method provides to start from a determined coordinates. ***(Code-6)***

➔ Map: *(M)*

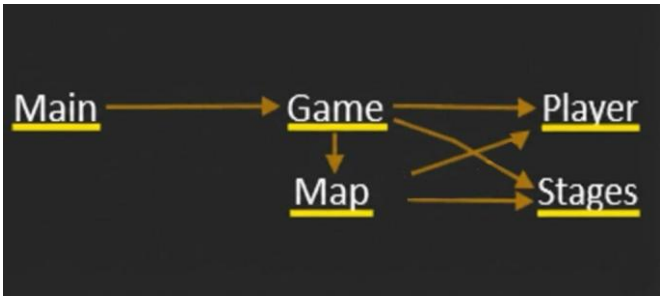
This class represents the game map for a single stage, including obstacles, spikes, doors, buttons, player interactions, and drawing the game display. The class accepts a stage object, a player object, and a game object in the constructor ***(Code-7)***. In this class' movePlayer method, player ***(P)*** and stage ***(S)*** objects are used to access some methods of player class as shown in the ***Code-8***. Additionally, collisions are handled in this class by using player object ***(P)*** again, and implementation is according to most penetrated side as shown in the ***Code-9*** by the help of some little methods. At the end of the class, map is drawn by the information in the data fields ***(Code-10***).

➔ **Game:** ***(G)***

This class accepts an arraylist which consists of stage objects ***(S)*** in its constructor as shown in ***Code-11***. The class is the core game manager by play method since this method arranges what will be shown in UI ***(Code-12)*** by use of player object ***(P).*** For instance, "end game banner" or "resetting game banner" may be shown ***(Figure-2)***. Also, mouse and keyboard inputs are handled in this class, and map object ***(M)*** is used while keyboard input method ***(Code-13)***. Player object ***(P)*** is used to be able to call respawn method while passing a stage as shown in ***Code-14***.

Figure-2: "Resetting the Game" banner

Interaction between classes:

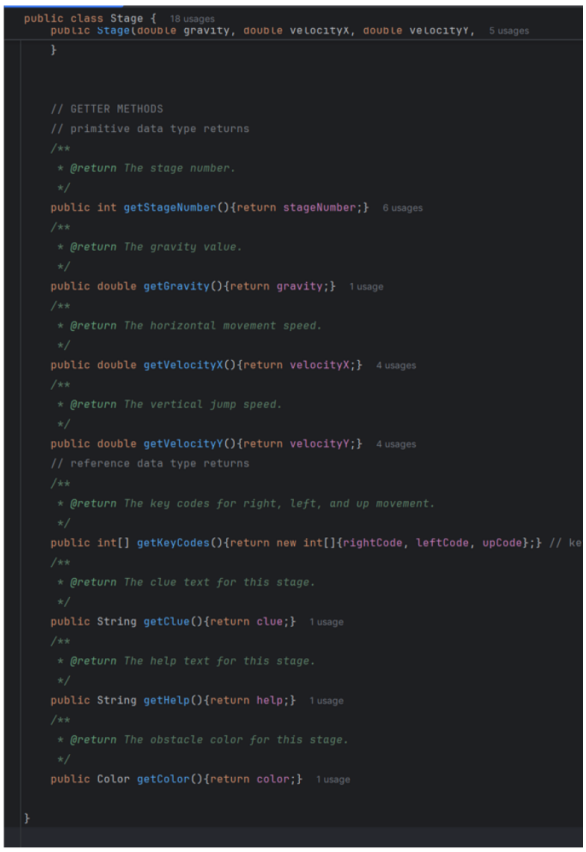


CODE SNIPPETS

```java
public class Main {

    /**
     * Initializes all stages and starts the game by calling the game object's method.
     * @param args command-line arguments (not used)
     */
    public static void main(String[] args){

        int nullButton = -1; // Cancel "up button" for stage 3

        // Given Stages
        // normal game
        Stage s1 = new Stage(-0.45, 3.65,10,0, KeyEvent.VK_RIGHT, KeyEvent.VK_LEFT, KeyEvent.VK_UP,"Arrow keys are required",
        // reversed buttons
        Stage s2 = new Stage(-0.45, 3.65,10,1, KeyEvent.VK_LEFT,  KeyEvent.VK_RIGHT,KeyEvent.VK_UP,"Not always straight forwar
        // bouncing
        Stage s3 = new Stage(-2,    3.65,24,2, KeyEvent.VK_RIGHT, KeyEvent.VK_LEFT, nullButton,    "A bit bouncy here",
        // multiple button presses
        Stage s4 = new Stage(-0.45, 3.65,10,3, KeyEvent.VK_RIGHT, KeyEvent.VK_LEFT, KeyEvent.VK_UP,"Never gonna give you up",
        // New stage (direction buttons are FTH)
        Stage s5 = new Stage(-0.45, 3.65,10,4, KeyEvent.VK_H,     KeyEvent.VK_F,    KeyEvent.VK_T,  "Center keyboarder",

        // Add the stages to the arraylist
        ArrayList<Stage> stages = new ArrayList<Stage>();
        stages.add(s1);
        stages.add(s2);
        stages.add(s3);
        stages.add(s4);
        stages.add(s5);

        // Start the game
        Game game = new Game(stages);
        game.play();

    }
}
```

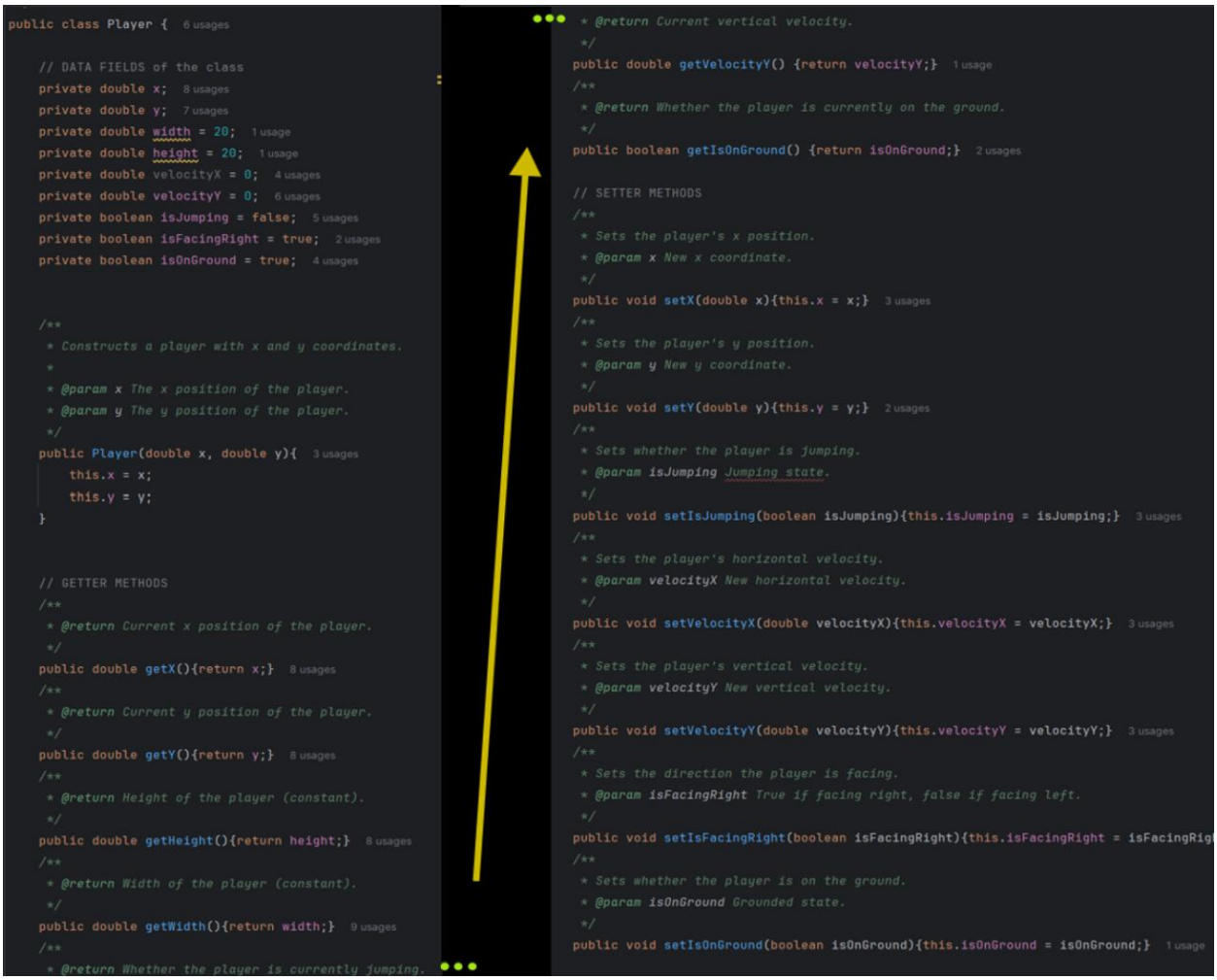
Code-1

```
public class Stage {  18 usages
    // DATA FIELDS of the class
    private int stageNumber;  2 usages
    private double gravity;  2 usages
    private double velocityX;  2 usages
    private double velocityY;  2 usages
    private int rightCode;  2 usages
    private int leftCode;  2 usages
    private int upCode;  2 usages
    private String clue;  2 usages
    private String help;  2 usages
    private Color color;  4 usages


    /**
     * Constructs a new Stage with the specified parameters.
     * Assigns obstacle color except white.
     *
     * @param gravity     The gravity value for the stage (affects jumping/falling).
     * @param velocityX   The horizontal movement speed.
     * @param velocityY   The vertical jump speed.
     * @param stageNumber The identifier number of the stage.
     * @param rightCode   The key code for moving right.
     * @param leftCode    The key code for moving left.
     * @param upCode      The key code for jumping (up).
     * @param clue        A short clue shown during gameplay.
     * @param help        Detailed help text shown when the help button is clicked.
     */
    public Stage(double gravity, double velocityX, double velocityY,  5 usages
                 int stageNumber, int rightCode, int leftCode,
                 int upCode, String clue, String help) {
        this.gravity = gravity;
        this.velocityX = velocityX;
        this.velocityY = velocityY;
        this.stageNumber = stageNumber;
        this.rightCode = rightCode;
        this.leftCode = leftCode;
        this.upCode = upCode;
        this.clue = clue;
        this.help = help;
        this.color = new Color((int)(Math.random()*256),
                               (int)(Math.random()*256),
                               (int)(Math.random()*256));
        // Prevent possibility of color being white
        while (color.equals(new Color( r: 255, g: 255, b: 255))){
            color = new Color((int)(Math.random()*256),
                              (int)(Math.random()*256),
                              (int)(Math.random()*256)
```



```
public class Stage {  18 usages
    public Stage(double gravity, double velocityX, double velocityY,  5 usages
    }



    // GETTER METHODS
    // primitive data type returns
    /**
     * @return The stage number.
     */
    public int getStageNumber(){return stageNumber;}  6 usages
    /**
     * @return The gravity value.
     */
    public double getGravity(){return gravity;}  1 usage
    /**
     * @return The horizontal movement speed.
     */
    public double getVelocityX(){return velocityX;}  4 usages
    /**
     * @return The vertical jump speed.
     */
    public double getVelocityY(){return velocityY;}  4 usages
    // reference data type returns
    /**
     * @return The key codes for right, left, and up movement.
     */
    public int[] getKeyCodes(){return new int[]{rightCode, leftCode, upCode};} // ke
    /**
     * @return The clue text for this stage.
     */
    public String getClue(){return clue;}  1 usage
    /**
     * @return The help text for this stage.
     */
    public String getHelp(){return help;}  1 usage
    /**
     * @return The obstacle color for this stage.
     */
    public Color getColor(){return color;}  1 usage

}
```

Code-2 & Code-3



```
public class Player {  6 usages

    // DATA FIELDS of the class
    private double x;  8 usages
    private double y;  7 usages
    private double width = 20;  1 usage
    private double height = 20;  1 usage
    private double velocityX = 0;  4 usages
    private double velocityY = 0;  6 usages
    private boolean isJumping = false;  5 usages
    private boolean isFacingRight = true;  2 usages
    private boolean isOnGround = true;  4 usages


    /**
     * Constructs a player with x and y coordinates.
     *
     * @param x The x position of the player.
     * @param y The y position of the player.
     */
    public Player(double x, double y){  3 usages
        this.x = x;
        this.y = y;
    }


    // GETTER METHODS
    /**
     * @return Current x position of the player.
     */
    public double getX(){return x;}  8 usages
    /**
     * @return Current y position of the player.
     */
    public double getY(){return y;}  8 usages
    /**
     * @return Height of the player (constant).
     */
    public double getHeight(){return height;}  8 usages
    /**
     * @return Width of the player (constant).
     */
    public double getWidth(){return width;}  9 usages
    /**
     * @return Whether the player is currently jumping.
     * @return Current vertical velocity.
     */
    public double getVelocityY() {return velocityY;}  1 usage
    /**
     * @return Whether the player is currently on the ground.
     */
    public boolean getIsOnGround() {return isOnGround;}  2 usages

    // SETTER METHODS
    /**
     * Sets the player's x position.
     * @param x New x coordinate.
     */
    public void setX(double x){this.x = x;}  3 usages
    /**
     * Sets the player's y position.
     * @param y New y coordinate.
     */
    public void setY(double y){this.y = y;}  2 usages
    /**
     * Sets whether the player is jumping.
     * @param isJumping Jumping state.
     */
    public void setIsJumping(boolean isJumping){this.isJumping = isJumping;}  3 usages
    /**
     * Sets the player's horizontal velocity.
     * @param velocityX New horizontal velocity.
     */
    public void setVelocityX(double velocityX){this.velocityX = velocityX;}  3 usages
    /**
     * Sets the player's vertical velocity.
     * @param velocityY New vertical velocity.
     */
    public void setVelocityY(double velocityY){this.velocityY = velocityY;}  3 usages
    /**
     * Sets the direction the player is facing.
     * @param isFacingRight True if facing right, false if facing left.
     */
    public void setIsFacingRight(boolean isFacingRight){this.isFacingRight = isFacingRig
    /**
     * Sets whether the player is on the ground.
     * @param isOnGround Grounded state.
     */
    public void setIsOnGround(boolean isOnGround){this.isOnGround = isOnGround;}  1 usage
```

Code-4

```java
/**
 * Makes the player jump with a given vertical velocity.
 *
 * @param jumpVelocity Velocity to apply when jumping.
 */
public void jump(double jumpVelocity) {  4 usages
    if (!isJumping) {
        velocityY = jumpVelocity;
        isJumping = true;
        isOnGround = false;
    }
}

/**
 * Moves the player to the left by a given speed.
 *
 * @param speedX Amount to move left.
 */
public void moveLeft(double speedX) {  2 usages
    velocityX = -speedX;
    x -= speedX;
}

/**
 * Moves the player to the right by a given speed.
 *
 * @param speedX Amount to move right.
 */
public void moveRight(double speedX) {  2 usages
    velocityX = speedX;
    x += speedX;
}

/**
 * Applies gravity to the player and updates vertical posi
 *
 * @param gravity The gravity acceleration value.
 */
public void applyGravity(double gravity) {  1 usage
    velocityY += gravity;
    y += velocityY;
}
```

```java
/**
 * Respawns the player at the given coordinates and resets movement states.
 *
 * @param spawnPoint Array containing x and y respawn coordinates.
 */
public void respawn(double[] spawnPoint){  2 usages
    this.x = spawnPoint[0];
    this.y = spawnPoint[1];
    this.velocityX = 0;
    this.velocityY = 0;
    isJumping = false;
    isOnGround = true;
}

/**
 * Draws the player on screen depending on the direction.
 */
public void draw(){  1 usage
    if (isFacingRight) {
        StdDraw.picture(x, y,  filename: "ElephantRight.png",  scaledWidth: 20,  scaledH
    } else {
        StdDraw.picture(x, y,  filename: "ElephantLeft.png",  scaledWidth: 20,  scaledHe
    }
}
```

Codes-5 & Codes-6

```java
/**
 * Constructs a map with the given stage and player reference.
 *
 * @param stage The current stage configuration.
 * @param player The player object.
 * @param game The game object.
 */
public Map(Stage stage, Player player, Game game){  4 usages
    this.stage = stage;
    this.player = player;
    this.game = game;
    // Store original button and door positions
    for (int i = 0; i < 4; i++) {
        originalButton[i] = button[i];
        originalDoor[i] = door[i];
    }
}
```

Code-7

```java
public void movePlayer(char direction){  3 usages

    if (direction == 'L') { // calls move left
        player.setIsFacingRight(false);
        player.moveLeft(stage.getVelocityX());
    } else if (direction == 'R') { // calls move right
        player.setIsFacingRight(true);
        player.moveRight(stage.getVelocityX());
    } else if (direction == 'U') { // calls jump
        player.jump(stage.getVelocityY());
    }

    // Bounce when on ground at stage 3
    if (stage.getStageNumber() == 2 && player.getIsOnGround()) {
        player.jump(stage.getVelocityY());
    }

}
```

Code-8

```java
private boolean checkObstacleCollision() {  1 usage
    boolean isOnGround = false;

    for (int[] obstacle : obstacles) {
        if (checkCollision(player.getX(), player.getY(), player.getWidth(), play
            double playerLeft = player.getX() - player.getWidth() / 2;
            double playerRight = player.getX() + player.getWidth() / 2;
            double playerTop = player.getY() + player.getHeight() / 2;
            double playerBottom = player.getY() - player.getHeight() / 2;

            double obstacleLeft = obstacle[0];
            double obstacleRight = obstacle[2];
            double obstacleTop = obstacle[3];
            double obstacleBottom = obstacle[1];

            // Calculate penetration depths to find which side is the true colli
            double leftPenetration = playerRight - obstacleLeft;
            double rightPenetration = obstacleRight - playerLeft;
            double topPenetration = obstacleTop - playerBottom;
            double bottomPenetration = playerTop - obstacleBottom;

            // Find minimum penetration
            double minPenetration = Math.min( Math.min(leftPenetration, rightPen

            // Prevent wrong collision detection due to high velocityY which ca
            if( stage.getStageNumber() == 2 && leftPenetration > 5 && rightPenet
                minPenetration = Math.min(bottomPenetration, topPenetration);
            }

            // Assign based on minimum penetration
            if (minPenetration == leftPenetration) {
                // collision from the left
                player.setX(obstacleLeft - player.getWidth() / 2);
                player.setVelocityX(0);
            } else if (minPenetration == rightPenetration) {
                // collision from the right
                player.setX(obstacleRight + player.getWidth() / 2);
                player.setVelocityX(0);
            } else if (minPenetration == topPenetration) {
                // collision from the top (player is below obstacle)
                player.setY(obstacleTop + player.getHeight() / 2);
                player.setVelocityY(0);
                isOnGround = true;
```

```java
/**
 * Renders the entire game map including player, pipes, spikes, door, button, and obst
 */
public void draw(){  1 usage

    // player
    player.draw();

    // obstacles
    StdDraw.setPenColor(stage.getColor());
    for (int[] obs : obstacles) {
        StdDraw.filledRectangle( x: (obs[0]+obs[2])/2,  y: (obs[1]+obs[3])/2,
                halfWidth: (obs[2]-obs[0])/2,  halfHeight: (obs[3]-obs[1])/2);
    }

    // spikes (according their locations)
    for (int[] spike : spikes) {...}

    // start pipe
    for (int[] pipe : startPipe) {...}

    // exit pipe
    for (int[] pipe : exitPipe) {...}

    // door
    StdDraw.setPenColor(StdDraw.GREEN);
    StdDraw.filledRectangle( x: (door[0] + door[2]) / 2.0,  y: (door[1] + door[3]) / 2
            halfWidth: (door[2] - door[0]) / 2.0,  halfHeight: (door[3] - door[1]) / 2.0);

    // button
    if (!isButtonPressing) {
        StdDraw.setPenColor(StdDraw.RED);
        StdDraw.filledRectangle( x: (button[0] + button[2]) / 2.0,  y: (button[1] + bu
                halfWidth: (button[2] - button[0]) / 2.0,  halfHeight: (button[3] - button
    }
    // button floor
    StdDraw.setPenColor(new Color( r: 22,  g: 22,  b: 100));
    StdDraw.filledRectangle( x: (buttonFloor[0] + buttonFloor[2]) / 2.0,  y: (buttonFl
            halfWidth: (buttonFloor[2] - buttonFloor[0]) / 2.0,  halfHeight: (buttonFloor[
}
```

Code-9 & Code-10

```java
public Game(ArrayList<Stage> stages) {  1 usage
    this.stages = stages;
}
```

Code-11

```java
public void play() {  1 usage
    // canvas size
    StdDraw.setCanvasSize( canvasWidth: 800,  canvasHeight: 600);
    StdDraw.setXscale(0, 800);
    StdDraw.setYscale(0, 600);

    // Create some objects by OOP
    Stage currentStage = getCurrentStage();
    player = new Player( x: 130,  y: 465);
    map = new Map(currentStage, player,  game: this);

    double lastTime = System.currentTimeMillis() / 1000.0;
    double passedStageTime = 0;
    boolean stagePassed = false;

    // main game loop
    StdDraw.enableDoubleBuffering();
    while (true) {

        // Arrange time
        double currentTime = System.currentTimeMillis() / 1000.0;
        double deltaTime = currentTime - lastTime;
        lastTime = currentTime;

        // Reset display
        if (isResetting) {
            resetMessageTime += deltaTime;

            // Draw reset message
            StdDraw.setPenColor(StdDraw.GREEN);
            StdDraw.filledRectangle( x: 400,  y: 275,  halfWidth: 400,  halfHeight: 75)
            StdDraw.setPenColor(StdDraw.WHITE);

            Font resetFont = new Font( name: "Arial", Font.BOLD,  size: 45);
            StdDraw.setFont(resetFont);
            StdDraw.text( x: 400,  y: 275,  text: "RESETTING THE GAME...");

            if (resetMessageTime >= 2.0) { // Wait to seconds
                isResetting = false;
                resetMessageTime = 0;
                resetGame();
                resetHelp();
                player = new Player( x: 130,  y: 465);
```

```java
/**
 * Handles user keyboard input for player movement.
 */
private void handleInput() {  1 usage
    if (StdDraw.isKeyPressed(getCurrentStage().getKeyCodes()[0])) {
        if (getCurrentStage().getStageNumber() == 1) {
            map.movePlayer( direction: 'R',  isReversedKeys: -1); // -1 f
        } else {
            map.movePlayer( direction: 'R');
        }
    }
    if (StdDraw.isKeyPressed(getCurrentStage().getKeyCodes()[1])) {
        if (getCurrentStage().getStageNumber() == 1) {
            map.movePlayer( direction: 'L',  isReversedKeys: -1); // -1 f
        } else {
            map.movePlayer( direction: 'L');
        }
    }
    if (StdDraw.isKeyPressed(getCurrentStage().getKeyCodes()[2])) {
        map.movePlayer( direction: 'U');
    }
}

/**
 * Increments the death counter by one.
 */
public void incrementDeath() {  1 usage
    deathNumber++;
}

/**
 * Updates the internal game timer.
 *
 * @param delta Time passed since last update.
 */
```

Code-12 & Code-13

```
// Passing stage banner
if (stagePassed) {
    passedStageTime += deltaTime;

    // Draw passing stage banner
    StdDraw.setPenColor(StdDraw.GREEN);
    StdDraw.filledRectangle( x: 400,  y: 275,  halfWidth: 400,  halfHeight: 75);
    StdDraw.setPenColor(StdDraw.WHITE);

    Font passFont = new Font( name: "Arial", Font.PLAIN,  size: 24);
    StdDraw.setFont(passFont);
    StdDraw.text( x: 400,  y: 290,  text: "You passed the stage");
    StdDraw.text( x: 400,  y: 260,  text: "But is the level over?!");

    if (passedStageTime >= 2.0) { // Wait 2 seconds
        stagePassed = false;
        passedStageTime = 0;
        stageIndex++;
        resetHelp();

        if (stageIndex >= stages.size()) {
            endGame();
            if (isFinished) {
                break;
            }
        } else {
            currentStage = getCurrentStage();
            player.respawn(new double[]{130, 465});
            map = new Map(currentStage, player,  game: this);
        }
    }
}
```

Code-14