Contents

Introduction

In the realm of classic browser-based games, **Chrome Dino** stands as a timeless favorite, captivating players with its simplicity and addictive gameplay. Drawing inspiration from this beloved title, I embarked on a journey to recreate its magic in a desktop environment using **Java Swing**. This project, aptly named "**Super Mario Runner**", merges the iconic elements of Chrome Dino with the charm of Super Mario Bros, offering players a delightful and nostalgic gaming experience.

Description

"**Super Mario Runner**" is a **Java Swing** utility game that pays homage to the retro gaming era by combining the gameplay mechanics of **Chrome Dino** with the beloved characters from **Super Mario Bros**. As players embark on this adventure, they step into the shoes of the iconic plumber, Mario, traversing a dynamically generated landscape filled with obstacles and challenges.

The game features intuitive controls, allowing players to guide Mario through the terrain with precision and agility. With each jump, players must navigate through a series of hurdles while striving to achieve the highest possible score.

To enhance the gaming experience, we incorporated vibrant graphics and lively animations, capturing the essence of the classic Super Mario universe. Additionally, players have the option to pause the game at any moment, providing flexibility and convenience during gameplay sessions.

"**Super Mario Runner**" offers endless entertainment, challenging players to test their reflexes and endurance as they embark on an exciting journey through obstacles. This Java Swing utility game ensures both excitement and nostalgia in equal measure.

App

The App class is the entry point of the game/application. It contains the **main()** method and initializes the game window.

```
SwingUtilities.invokeLater(() -> new GameWindow());
```

1. The **Main** Method or Entry Point:

- The **main()** method is where the program starts its execution. It serves as the **entry point** of the game/application.

2. Initializing the Game Window:
   - **SwingUtilities.invokeLater()** is a method used to ensure that GUI-related tasks are executed safely on the AWT event dispatching thread.
   - **invokeLater()** takes a Runnable object, which represents a task to be executed.
   - In this case, the task is creating a new instance of the **GameWindow** class, which likely sets up and displays the game interface.
3. Asynchronous Execution:
   - By using **invokeLater()**, the creation of the game window is handled asynchronously. This means it doesn't block the main thread of execution.
   - Asynchronous execution ensures that the user interface remains responsive while the game window is being initialized.

Interfaces

Game Settings

This **GameSettings** interface defines constants settings for various aspects of a game such as window properties, game physics, ground and obstacle parameters, Mario's attributes, background, environments, and score settings. It also includes some general options and a constant for debugging.

Drawable

The **Drawable** interface extends the **GameSettings** interface and defines three methods: **update()**, **draw(Graphics g)**, and **reset()**.

```
public interface Drawable extends GameSettings {
    void update(); // This method is used to update the state of the object.
    void draw(Graphics g); // This method is used to draw the object on a
Graphics context.
    void reset(); // reset the state of the object to its initial state
}
```

The **Drawable** interface is often used in game development to define an object that can be drawn on the screen and updated periodically.

Sound Manager

It defines an abstract method that will manage audio playback.

```
    void toggleMic();
```

Components

Backgrounds

This **Background** class implements the **Drawable** interface and represents the background.

```
    static {
        CLOUD_IMAGES = new BufferedImage[CLOUD_LEVEL + 1]; // +1 for a safe
memory allocation...
        for (int i = 0; i <= CLOUD_LEVEL; i++) { // Load all cloud images into
the array
            CLOUD_IMAGES[i] = new Resource().getResourceImage("/cloud/cloud_" + i
+ ".png");
        }
        System.out.println("Cloud level: " + CLOUD_LEVEL);
    }
```

A static block that loads all cloud images into an array **CLOUD_IMAGES**.

```
    /**
     * Set the default color and fill the entire window with it.
     */
    public Background() {
        backgroundColor = BackgroundColors.DEFAULT;
        backgroundInit();
    }
```

The constructor sets the default background color and initializes the background.

```
    /**
     * Initializes the background by generating a specified number of cloud
images at random positions within the window.
     *
     * @param None
     * @return None
     */
```

```
    private void backgroundInit() {
        cloudImages = new ArrayList<>();
        Random random = new Random();

        System.out.print("Cloud density: ");
        for (int i = 0; i <= cloud_density; i++) {
            int z = (int) (Math.random() * WINDOW_HEIGHT);
            System.out.print(z + " ");
            int y = random.nextInt(WINDOW_HEIGHT - z); // Random y within the
window height
            int x = random.nextInt(WINDOW_WIDTH); // Random x within the window
width
            BufferedImage cloudImage =
CLOUD_IMAGES[random.nextInt(CLOUD_IMAGES.length)]; // Random cloud image
            ComponentImage cloud = new ComponentImage(cloudImage, x, y,
Color.WHITE);
            cloudImages.add(cloud);
        }
        System.out.println();
    }
```

This **backgroundInit()** method initializes the background by generating a specified number of cloud images at random positions within the window.

```
    /**
     * Updates the cloud images by moving them horizontally and resetting their
     * position if they move off the screen.
     */
    @Override
    public void update() {
        for (ComponentImage cloud : cloudImages) {
            cloud.x -= BACKGROUND_SPEED;
            if (cloud.x <= -cloud.image.getWidth()) {
                cloud.x = WINDOW_WIDTH;
            }
        }
    }



    /**
     * Draws the background based on the specified background color.
     *
     * @param g the Graphics object used for drawing
```

```
     */
    @Override
    public void draw(Graphics g) {

        switch (backgroundColor) {
            case DEFAULT:
                defaultColor(g);
                break;
            case DARK:
                g.setColor(Color.BLACK);
                g.fillRect(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT);
                break;
        }

        for (ComponentImage clouds : cloudImages) {
            if (debugMode) {
                g.setColor(clouds.debugColor);
                g.drawRect(clouds.x, clouds.y, clouds.image.getWidth(),
clouds.image.getHeight());
            }
            g.drawImage(clouds.image, clouds.x, clouds.y, null);
        }
    }
```

draw(Graphics g) method draws the background based on the specified (Game Settings interface) background color.

```
    /**
     * Reset the object by initializing the background and setting the background
     * color to default.
     */
    @Override
    public void reset() {
        backgroundInit();
        backgroundColor = BackgroundColors.DEFAULT;
    }
```

Ground

The **Ground** class represent the ground of this game. It implements the **Drawable** interface, which ensure that it have to draw on a graphics object. It has two static **ComponentImage** objects, **FIRST_GROUND** and **SECOND_GROUND**, which

are initialized with an image and a color. These represent the first and second parts of the ground image respectively.

```
    private static final ComponentImage FIRST_GROUND = new
ComponentImage(GROUND_IMAGE, 0, Color.green);
    private static final ComponentImage SECOND_GROUND = new
ComponentImage(GROUND_IMAGE, GROUND_IMAGE.getWidth(),
            Color.blue);
    public Ground() {
        groundInit();
    }
```

The **Ground** constructor calls the **groundInit();** method to initialize the **groundImages** ArrayList with these two ground images.

```
    /**
     * Initializes the ground by creating a new ArrayList and adding
     * the ground images to it. Also initializes the first and
     * second ground images.
     */
    private void groundInit() {
        groundImages = new ArrayList<>();

        groundImages.add(FIRST_GROUND);
        groundImages.add(SECOND_GROUND);
    }
```

The **groundInit**() method creates an ArrayList to hold the ground images.

```
    /**
     * @see ComponentImage#x Defines also inequalities arising
     *      from updating x before changing GroundImage position (I think)
     * At this point, I realize that GPT is an assho;e.
     */
    public void update() {
        FIRST_GROUND.x -= GamePanel.gameSpeed;
        SECOND_GROUND.x -= GamePanel.gameSpeed;

        if (FIRST_GROUND.x <= -FIRST_GROUND.image.getWidth()) {
            FIRST_GROUND.x = SECOND_GROUND.image.getWidth() + SECOND_GROUND.x;
        }
        if (SECOND_GROUND.x <= -SECOND_GROUND.image.getWidth()) {
            SECOND_GROUND.x = FIRST_GROUND.image.getWidth() + FIRST_GROUND.x;
        }
```

```
    }
```

The **update**() method moves the ground images horizontally by a certain speed defined in **GamePanel.gameSpeed**. If a ground image goes off the screen, it is moved back to the other side of the screen.

```
    /**
     * Draws the ground images on the graphics object.
     *
     * @param g the graphics object on which to draw
     * @return void
     */
    public void draw(Graphics g) {
        for (ComponentImage ground : groundImages) {
            if (GamePanel.debugMode) {
                g.setColor(ground.debugColor);
                g.drawLine(ground.x, GROUND_Y, ground.image.getWidth() +
ground.x, GROUND_Y);
            }
            g.drawImage(ground.image, ground.x, GROUND_Y + ground_y_image_offset,
null);
        }
    }
```

This method iterates over the **groundImages** ArrayList and draws each one on the graphics object.

Obstacles

ObstacleImage

ObstacleImage class create obstacles image within the window.

- **ObstacleImage(BufferedImage OBSTACLE_IMAGE)** initializes the obstacle image with default values.
- **ObstacleImage(BufferedImage OBSTACLE_IMAGE, int spaceBehind)** initializes the obstacle image with a specific space behind value.
- **ObstacleImage(BufferedImage OBSTACLE_IMAGE, int x, int spaceBehind)** initializes the obstacle image with specific x-coordinate and space behind values.

```
    public ObstacleImage(BufferedImage OBSTACLE_IMAGE) {
        this.OBSTACLE_IMAGE = new ComponentImage(OBSTACLE_IMAGE, 0, GROUND_Y -
OBSTACLE_IMAGE.getHeight(), Color.red);
        this.spaceBehind = 0;

        coordinates = new Coordinates(this.OBSTACLE_IMAGE.x,
                this.OBSTACLE_IMAGE.y,
                OBSTACLE_IMAGE.getWidth(),
                OBSTACLE_IMAGE.getHeight());
    }

    public ObstacleImage(BufferedImage OBSTACLE_IMAGE, int spaceBehind) {
        this.OBSTACLE_IMAGE = new ComponentImage(OBSTACLE_IMAGE, 0, GROUND_Y -
OBSTACLE_IMAGE.getHeight(), Color.red);
        this.spaceBehind = spaceBehind;

        coordinates = new Coordinates(this.OBSTACLE_IMAGE.x,
                this.OBSTACLE_IMAGE.y,
                OBSTACLE_IMAGE.getWidth(),
                OBSTACLE_IMAGE.getHeight());
    }

    public ObstacleImage(BufferedImage OBSTACLE_IMAGE, int x, int spaceBehind) {
        this.OBSTACLE_IMAGE = new ComponentImage(OBSTACLE_IMAGE, x, GROUND_Y -
OBSTACLE_IMAGE.getHeight(), Color.red);
        this.spaceBehind = spaceBehind;

        coordinates = new Coordinates(this.OBSTACLE_IMAGE.x,
                this.OBSTACLE_IMAGE.y,
                OBSTACLE_IMAGE.getWidth(),
                OBSTACLE_IMAGE.getHeight());
    }


    /**
     * Set the space behind value.
     *
     * @param spaceBehind the new space behind value to set
     */
    public void setSpaceBehind(int spaceBehind) {
        this.spaceBehind = spaceBehind;
    }

    /**
     * Set the x-coordinate for the obstacle image and coordinates.
```

```
     *
     * @param x the new x-coordinate value
     */
    public void setX(int x) {
        this.OBSTACLE_IMAGE.x = x;
        coordinates.x = x;
    }
```

Obstacles

The **Obstacles** class is responsible for managing the obstacles that Mario has to overcome.

- **private static final int RANGE_SPACE_BETWEEN_OBSTACLES** and **private static final ArrayList<ObstacleImage> OBSTACLE_IMAGES** are constants and lists used by the class. The first constant is the range of space between obstacles, and the second list contains all the images for the obstacles.
- **public Obstacles()** initializes the obstacle images and sets up the first set of obstacles.

```
    /**
     * Initializes the first set of obstacles.
     */
    private void initFirstObstacles() {
        incomingObstacles = new ArrayList<>();

        for (int i = 0; i < max_incoming_obstacles; i++) {
            ObstacleImage rand = getRandomObstacle();

            incomingObstacles.add(rand);
            if (i == 0) {
                incomingObstacles.get(0).setX(OBSTACLES_FIRST_OBSTACLE_X);
            } else {
                incomingObstacles.get(i)
                        .setX(incomingObstacles.get(i - 1).getX() +
incomingObstacles.get(i - 1).getSpaceBehind());
            }
        }
    }

    /**
     * Generates a random obstacle image.
     *
     * @return a new ObstacleImage with a randomly selected image and space
```

```
     */
    private ObstacleImage getRandomObstacle() {
        int randCactus = (int) (Math.random() * (OBSTACLE_IMAGES.size()));
        ObstacleImage randObstacle = OBSTACLE_IMAGES.get(randCactus);

        return new ObstacleImage(randObstacle.getOBSTACLE_IMAGE(),
getRandomSpace());
    }

    /**
     * Generate a random obstacle image at the given x coordinate.
     *
     * @param x the x coordinate for the obstacle image
     * @return the randomly generated obstacle image
     */
    private ObstacleImage getRandomObstacle(int x) {
        int randCactus = (int) (Math.random() * (OBSTACLE_IMAGES.size()));
        ObstacleImage randObstacle = OBSTACLE_IMAGES.get(randCactus);

        return new ObstacleImage(randObstacle.getOBSTACLE_IMAGE(), x,
getRandomSpace());
    }

    /**
     * A method to return a random space between obstacles.
     *
     * @return the random space between obstacles
     */
    private int getRandomSpace() {
        return (int) (Math.random() * RANGE_SPACE_BETWEEN_OBSTACLES) +
OBSTACLES_MIN_SPACE_BETWEEN;
    }


    /**
     * Checks if there is a collision between Mario and any incoming obstacles.
     *
     * @return true if there is a collision, false otherwise
     */
    public boolean isCollision() {
        for (ObstacleImage obstacle : incomingObstacles) {
            for (Coordinates marioCoordinates : Mario.constructedCoordinates)
                if (marioCoordinates.intersects(obstacle.coordinates)) {
                    return true;
                }
```

```
        }
        return false;
    }


    /**
     * Updates the position of the incoming obstacles and generates new obstacles
     * when the first one goes off the screen.
     *
     * This function iterates over each obstacle in the incomingObstacles list
and
     * updates its x-coordinate by subtracting the game speed.
     * If the first obstacle goes off the screen (i.e., its x-coordinate is less
     * than - its width), a new obstacle is generated.
     * The last obstacle in the list is used to determine the position and space
     * behind the new obstacle.
     * The first obstacle in the list is removed from the list, and a new
obstacle
     * is added at the end of the list with a random position and space behind.
     *
     * @return void
     */
    @Override
    public void update() {
        for (ObstacleImage obstacle : incomingObstacles) {
            obstacle.setX(obstacle.getX() - GamePanel.gameSpeed);
        }

        if (incomingObstacles.get(0).getX() < -
incomingObstacles.get(0).getOBSTACLE_IMAGE().getWidth()) {
            ObstacleImage lastIncomingObstacle =
incomingObstacles.get(incomingObstacles.size() - 1);

            incomingObstacles.remove(0);
            incomingObstacles
                    .add(getRandomObstacle(lastIncomingObstacle.getX() +
lastIncomingObstacle.getSpaceBehind()));
            incomingObstacles.get(incomingObstacles.size() -
1).setSpaceBehind(getRandomSpace());
        }
    }

    /**
     * Draw existing obstacles on the graphics object. Check for collision with
     * Mario and remove obstacles that are out of the screen.
```

```
     *
     * @param g the Graphics object to draw on
     * @return void
     */
    @Override
    public void draw(Graphics g) {
        // Draw existing obstacles
        for (int i = 0; i < incomingObstacles.size(); i++) {
            ObstacleImage obstacle = incomingObstacles.get(i);
            if (GamePanel.debugMode) {
                g.setColor(obstacle.getDebugColor());
                g.drawRect(obstacle.coordinates.x, obstacle.coordinates.y,
obstacle.coordinates.width,
                        obstacle.coordinates.height);
            }
            g.drawImage(obstacle.getOBSTACLE_IMAGE(), obstacle.getX(),
obstacle.getY(), null);

            // If the obstacle is out of the screen, remove it and add a new one
            if (obstacle.getX() < -obstacle.getOBSTACLE_IMAGE().getWidth()) {
                incomingObstacles.remove(i);
                i--; // Decrement i because we removed an obstacle
                ObstacleImage lastIncomingObstacle =
incomingObstacles.get(incomingObstacles.size() - 1);
                incomingObstacles
                        .add(getRandomObstacle(lastIncomingObstacle.getX() +
lastIncomingObstacle.getSpaceBehind()));
                incomingObstacles.get(incomingObstacles.size() -
1).setSpaceBehind(getRandomSpace());
            }
        }
    }
```

Mario

MarioStates

Defines an enumeration of possible states that Mario can be in.

```
/*
 * Enum class for Mario's states
 *
 * IDLE: The character is standing
 * RUNNING: The character is running
 * JUMPING: The character is jumping
```

```
 * FALL: The character is falling
 * DIE: The character is dead
 */
public enum MarioStates {
    IDLE, RUNNING, JUMPING, FALL, DIE
}
```

Mario

This class represents the **Mario** character in the game. It implements the **Drawable** and **SoundManager** interfaces.

```
    /**
     * Collision adjustments.
     * It's a modified version of bohdankordon/chrome-dino-java
     * ------------------------------------------------------------------------
     * ------------------------------------------------------------------------
     *
     * @see https://github.com/bohdankordon/chrome-dino-
java/blob/966e24fd7a85d9749056182d815503a60ddd2231/src/game_object/Dino.java
     */
    public static ArrayList<Coordinates> constructedCoordinates = new
ArrayList<>();
    private static final Coordinates collisionLeft = new Coordinates(0, 15, 11 -
MARIO_BORDER_SIZE,
            21 - MARIO_BORDER_SIZE);
    private static final Coordinates collisionMiddle = new Coordinates(10, 0, 22
- MARIO_BORDER_SIZE,
            45 - MARIO_BORDER_SIZE);
    private static final Coordinates collisionRight = new Coordinates(31, 0, 10 -
MARIO_BORDER_SIZE,
            21 - MARIO_BORDER_SIZE);



    /**
     * Constructor for the Mario class.
     *
     * Initializes the Mario object with default animations
     * and collision coordinates.
     *
     * @see MarioStates // @RUNNING Animation objects...
     */
    public Mario() {
```

```
        for (int i = 0; i <= MARIO_FRAME; i++) {
            runAnimation.addFrame(ResizeImage.getResizedImage("/marioFrame/frame_
" + i + ".png", 50));
        }
        System.out.println("MArio frame size: " + MARIO_FRAME);
        // Add more @Animation addFrame

        // Initialize collision coordinates
        constructedCoordinates.add(
                new Coordinates((int) X, (int) y + collisionLeft.y,
collisionLeft.width, collisionLeft.height));
        constructedCoordinates.add(
                new Coordinates((int) X + collisionMiddle.x, (int) y,
collisionMiddle.width, collisionMiddle.height));
        constructedCoordinates.add(
                new Coordinates((int) X + collisionRight.x, (int) y,
collisionRight.width, collisionRight.height));
    }



    /**
     * This function is responsible for making Mario jump in the game.
     *
     * It checks the current state of Mario and transitions him to the jumping
state
     * if he is running. It also updates the vertical speed and position of Mario
     * to simulate the jump. Additionally, it stops any ongoing jump sound and
plays
     * the jump sound if it is not muted.
     *
     * If Mario is not running and is already in the air, it sets the jump
requested
     * flag to true.
     *
     * @param None
     * @return None
     */
    public void jump() {
        if (marioState == MarioStates.RUNNING) {
            marioState = MarioStates.JUMPING;

            speedY = -MARIO_JUMP_STRENGTH;
            y += speedY;
```

```
            // It prevents from layering sounds and game freeze
            if (!jumpSound.isNull() && !isMuted) {
                if (jumpSound.isOpen())
                    jumpSound.stop();
            }
            if (!isMuted) {
                jumpSound.play();
            }
        } else if (isInAir()) {
            jumpRequested = true;
        }
    }



    /**
     * Draw state into Graphics object
     *
     * @param g Graphics object used for drawing
     */
    @Override
    public void draw(Graphics g) {
        if (debugMode) {
            for (Coordinates coordinates : constructedCoordinates) {
                g.setColor(Color.BLACK);
                g.drawRect(coordinates.x, coordinates.y, coordinates.width,
coordinates.height);
            }
        }
        switch (marioState) {
            case IDLE:
                g.drawImage(idleImage, (int) X, (int) y, null);
                break;
            case JUMPING:
                g.drawImage(jumpImage, (int) X, (int) y, null);
                break;
            case FALL:
                g.drawImage(fallImage, (int) X, (int) y, null);
                break;
            case RUNNING:
                runAnimation.update();
                g.drawImage(runAnimation.getFrame(), (int) X, (int) y, null);
                break;
```

```
            case DIE:
                g.drawImage(dieImage, (int) X, (int) y, null);
                break;
            default:
                break;
        }
    }
```

Utility

Animation

The **Animation** class is a utility class that manages a list of images (frames) and provides methods to update and retrieve the current frame. It has the following characteristics:

- It has a private **DeltaTime** object, which is used to control the timing of the animation.
- It has a private **ArrayList** of **BufferedImage** objects, which represents the frames of the animation.
- Variable **index**, which keeps track of the current frame.

```
/**
 * Constructs an Animation object with a given frame delay in milliseconds.
 * The delay is the amount of time between each frame in the animation.
 *
 * @param deltaTime the frame delay in milliseconds
 */
public Animation(int deltaTime) {
    frames = new ArrayList<>();
    index = 0;

    this.DELTA_TIME = new DeltaTime(deltaTime);
}


/**
 * This method is called to update the state of the object. It checks if the
 * delta time is available to execute the update. If it is, it increments the
 * index and resets it to 0 if it exceeds the size of the frames array.
 *
 * @param None
 * @return None
```

```
     */
    public void update() {
        if (DELTA_TIME.canExecute()) {
            index++;
            if (index >= frames.size()) {
                index = 0;
            }
        }
    }
```

DeltaTime

This class is used to control the frame rate. It ensures that a certain amount of time (delta time) has passed before each frame update. The **DELTA_TIME** is the minimum time that should pass between frames.

```
    public DeltaTime(int deltaTime) {
        this.DELTA_TIME = deltaTime;
    }
```

DeltaTime(int deltaTime) is the constructor of that class. It takes an integer argument, **deltaTime**, which sets the minimum time that should pass between frames. This value is stored in the **DELTA_TIME** variable.

```
    /**
     * Checks if the delta time is available to execute the update. If it is, it
     * returns true, otherwise it returns false.
     *
     * @return
     */
    public boolean canExecute() {
        if (System.currentTimeMillis() - lastTime > DELTA_TIME) {
            lastTime = System.currentTimeMillis();
            return true;
        }
        return false;
    }
```

Sound

The **Sound** class is a utility class that handles audio playback. It implements the **SoundManager** interface.

```
    public Sound(String fileName) {
        // Construct the file path relative to the project's root directory
```

```
        this.FILE = new File("lib/audio/" + fileName);
        // Print out the file path for debugging purposes
        System.out.println("Audio: " + fileName);
    }
```

play() method plays the audio file. It initializes the audio input stream and clip, sets up a line event listener to handle the clip's stop event, opens the clip with the audio input stream, and starts the clip.

```
    /**
     * Plays the audio file.
     *
     * This function initializes the audio input stream and clip, sets up a line
     * event listener to handle the clip's stop event,
     * opens the clip with the audio input stream, and starts the clip.
     *
     * @throws Exception if there is an error in initializing the audio input
stream
     *                   or clip.
     */
    public void play() {
        try {
            ais = AudioSystem.getAudioInputStream(FILE);
            clip = AudioSystem.getClip();
            event = event -> {
                if (event.getType() == LineEvent.Type.STOP)
                    clip.close();
            };
            clip.addLineListener(event);
            clip.open(ais);
            clip.start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

stop() method stops the audio playback. It removes the line listener and stops the clip.

```
    /**
     * It needs to be done before playing song
     *
     * It is important to remove line listener here, because
     * LineListener will try to execute code which can freeze the game
```

```
     * if someone will rapid click buttons in longer sounds
     */
    public void stop() {
        clip.removeLineListener(event);
        clip.stop();
    }
```

playInLoop() attempts to get an audio input stream (ais) from the specified audio file (FILE).

```
    /**
     * A method to play the audio file in a continuous loop.
     */
    public void playInLoop() {
        try {
            ais = AudioSystem.getAudioInputStream(FILE);
            clip = AudioSystem.getClip();
            clip.open(ais);
            clip.loop(Clip.LOOP_CONTINUOUSLY);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Resource

```
    /**
     * Retrieves a BufferedImage from the specified file path relative to the
"lib"
     * folder.
     *
     * @param path the relative path of the image file
     * @return the BufferedImage object read from the file, or null if the file
does
     *         not exist or an I/O error occurs
     */
    public BufferedImage getResourceImage(String path) {
        try {
            // Constructing the file path relative to the "lib" folder
            File file = new File("lib", path);
            // Check if the file exists before attempting to read it
            if (!file.exists()) {
                System.err.println("Image file not found: " +
file.getAbsolutePath());
```

```
                return null;
            }
            BufferedImage img = ImageIO.read(file);
            System.out.println("Image: " + path);
            return img;
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
```

getResourceImage(String path) method takes a String parameter path representing the relative path of the image file. It attempts to read the image file from the specified path and returns the **BufferedImage** object if successful. If the file does not exist or an I/O error occurs, it returns null.

Resize Image

This class provides a static method, **getResizedImage()**, to resize an image to a specified height. Unlike the **getResourceImage(String path)** method in the **Resource** class, this **getResizedImage(String path, int height)** method takes an additional parameter, **int height**, to resize a resource (image) to a specific height.

UI

Intro

This class **Intro** implements the **Drawable** interface. It is used to display an intro image and play the intro music.

```
    public Intro() {
        introLabel.setBounds((WINDOW_WIDTH - image.getWidth()) / 2,
                (WINDOW_HEIGHT - image.getHeight()) / 2 - 50,
                image.getWidth(), image.getHeight());

        overworld.playInLoop(); // play the intro music
    }
```

It initializes the **Intro** object by setting the bounds of the **introLabel** and playing the **intro** music.

```
    /**
     * Draws an image on the graphics context at the center of the window.
     *
     * @param g the graphics context to draw on
     */
    @Override
    public void draw(Graphics g) {
        g.drawImage(image, (WINDOW_WIDTH - image.getWidth()) / 2, (WINDOW_HEIGHT
- image.getHeight()) / 2 - 50, null);
    }
```

Score

This class represents the scoreboard of the game, handling the score system. It implements the Drawable and SoundManager interfaces, suggesting it's responsible for drawing the score and managing audio related to the score.

```
    /**
     * Builds a score string with leading zeros.
     *
     * @param score the score to be formatted
     * @return the formatted score string
     */
    private String scoreBuilder(int score) {
        StringBuilder ret = new StringBuilder(Integer.toString(score));
        char zero = '0';

        for (int i = 0; i < SCORE_MAX_ZEROS - Integer.toString(score).length();
i++) {
            ret.insert(0, zero);
        }

        return ret.toString();
    }


    /**
     * Plays a sound if the score is a multiple of 100 and the sound has not been
     * played yet.
     *
     * @return void
     */
    private void playSound() {
        if (score > 0 && score % 100 == 0 && !isPlayed && !isMuted) {
            isPlayed = true;
```

```
            SCORE_SOUND.play();
        }
    }


    /**
     * Checks if the current score is higher than the high score.
     *
     * @return true if the current score is higher than the high score, false
     *         otherwise
     */
    private boolean isHighScore() {
        return highScore < score;
    }


    /**
     * Reads the high score from a file if it exists, and returns it as an
integer.
     *
     * @return the high score as an integer
     */
    private static int readHighScore() {
        long highScore = 0;
        if (checkFileExistence()) {
            try {
                BufferedReader reader = new BufferedReader(new
FileReader(SCORE_FILE_NAME));
                highScore = Long.parseLong(reader.readLine());
                reader.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Score system: High score read");
        System.out.println("High score: " + highScore);
        return (int) highScore;
    }


    /**
     * https://stackoverflow.com/questions/12350248/java-difference-between-
filewriter-and-bufferedwriter
     * ------------------------------------------------------------------------
     * BufferedWriter is more efficient if you
     * - have multiple writes between flush/close
```

```
     * - ! the writes are small compared with the buffer size.
     * ------------------------------------------------------------------------
     * BufferedWriter is more efficient. It saves up small writes and writes in
one
     * larger chunk if memory
     * BufferedWriter. Calling
     * write calls to the OS which is slow so having as few writes as possible is
     * usually desirable.
     * ------------------------------------------------------------------------
     */
    public void writeHighScore() {
        if (isHighScore()) {
            try {
                BufferedWriter writer = new BufferedWriter(new
FileWriter(SCORE_FILE_NAME));
                writer.write(Integer.toString(score));
                writer.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
            highScore = score;
            System.out.println("Score system: New high score (" + highScore +
")");
        }
    }
    /**
     * Draws the score and high score on the graphics object.
     *
     * @param g the graphics object on which to draw
     */
    @Override
    public void draw(Graphics g) {
        g.setColor(Color.BLACK);
        g.setFont(new Font("Consolas", Font.BOLD, 18));
        g.drawString(printScore(score), WINDOW_WIDTH - 100, 40);
        g.setColor(Color.GRAY);
        g.drawString("HI " + printScore(highScore), WINDOW_WIDTH - 200, 40);
    }
```

Pause

It is used to display a "PAUSED" (image) in the center of a window when the game is **paused**.

```
    // constants for the dimensions of the PAUSED_TEXT image
```

```
    private static final BufferedImage PAUSED_TEXT = new
Resource().getResourceImage("/Paused.png");


    /**
     * Draws the PAUSED_TEXT image on the graphics object at the center of the
     * window.
     *
     * @param g the graphics object to draw on
     * @return void
     */
    @Override
    public void draw(Graphics g) {
        g.drawImage(PAUSED_TEXT, (WINDOW_WIDTH - PAUSED_TEXT.getWidth()) / 2,
                (WINDOW_HEIGHT - PAUSED_TEXT.getHeight()) / 2 - 70, null);
    }
```

Game Over

It just draw **game over** and **restart** image in the window.

```
    private static final BufferedImage TEXT = new
Resource().getResourceImage("/Game-over.png");
    private static final BufferedImage RESTART_BUTTON = new
Resource().getResourceImage("/Restart.png");


    /**
     * Draws the game over screen with the given graphics object.
     *
     * @param g the graphics object used for drawing
     */
    @Override
    public void draw(Graphics g) {
        g.drawImage(TEXT, (WINDOW_WIDTH - TEXT.getWidth()) / 2, (WINDOW_HEIGHT -
TEXT.getHeight()) / 2 - 70, null);
        g.drawImage(RESTART_BUTTON, (WINDOW_WIDTH - RESTART_BUTTON.getWidth()) /
2,
                (WINDOW_HEIGHT - RESTART_BUTTON.getHeight()) / 2 - 30, null);
    }
```

isMute

```
    /**
     * Draws the mute icon on the graphics object.
     *
     * Check if the sound is on or off
```

```
     *
     * @see GamePanel#paintComponent(Graphics g)
     *
     * @param g the graphics object to draw on
     * @return void
     */
    public void draw(Graphics g) {
        g.drawImage(MUTE_IMAGE, 5, 5, null);
    }
```

Main

Game Panel

This is a class definition for a game panel. Here's a succinct explanation of each class method:

```
    public GamePanel() {
        setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
        setLayout(null);
        setVisible(true);

        add(introUI.introLabel);

        mainThread.start();
    }
```

This constructor sets the size of the panel, sets the layout to null, sets the panel to be visible, and adds the intro label to the panel. It also starts a new thread to run the game loop.

```
    /**
     * Starts the game by setting the running flag to true, stopping the intro
     * music, and hiding the intro UI.
     *
     * @param None
     * @return None
     */
    public void startGame() {
        System.out.println("\nGame log");
        System.out.println("-------------------------------------------------");

        running = true;
```

```
        intro = false;
        introUI.overworld.stop(); // stop the intro music
        if (running == true) {
            System.out.println("Running...");
        }
    }


    /**
     * Resets the game state and starts a new game.
     *
     * This function sets the gameOver flag to false, the running flag to true,
     * and resets the game speed to the initial value. It also resets the score,
     * mario, obstacles, ground, and background objects. Finally, it starts a new
     * thread to run the game loop.
     *
     * @param None
     * @return None
     */
    public void resetGame() {
        gameOver = false;
        running = true;

        gameSpeed = game_start_speed;

        scoreUI.reset();
        mario.reset();
        obstacles.reset();
        ground.reset();
        background.reset();
        mainThread = new Thread(this);
        mainThread.start();
    }
```

public void paintComponent(Graphics g) method paints the components using the provided Graphics object.

```
    /**
     * MAIN PAINT METHOD
     * ------------------------------------------------------
     *
     * Paints the components using the provided Graphics object.
     *
```

```
     * @param g the Graphics object to paint with
     * @return void
     */
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        background.draw(g);

        if (isMuted) {
            mute.draw(g);
        }

        if (paused)
            pausedUI.draw(g);
        if (gameOver)
            gameOverUI.draw(g);
        if (!intro)
            scoreUI.draw(g);

        ground.draw(g);
        mario.draw(g);
        obstacles.draw(g);

        if (intro)
            introUI.draw(g);
    }
```

public void run() method is the main game loop. It first runs the intro loop, then the main game loop. In the main game loop, it handles game timing, game logic, and rendering output.

```
    @Override
    public void run() {
        // INTRO LOOP FOR EASTER EGG
        while (intro) {
            try {
                int msPerFrame = 1000 / game_fps;
                Thread.sleep(msPerFrame);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            repaint();
        }
```

```
        // MAIN GAME LOOP
        while (running) {
            // GAME TIMING
            try {
                int msPerFrame = 1000 / game_fps;
                Thread.sleep(msPerFrame);
                if (paused) {
                    synchronized (PAUSE_LOCK) {
                        repaint();
                        PAUSE_LOCK.wait();
                    }
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            // GAME LOGIC
            changeGameSpeed();
            scoreUI.update();
            background.update();
            mario.update();
            ground.update();
            obstacles.update();

            if (obstacles.isCollision()) {
                mario.die();
                if (Mario.isMario)
                    introUI.overworld.stop();
                scoreUI.writeHighScore();
                gameOver = true;
                running = false;
                System.out.println("Game over");
            }
            // RENDER OUTPUT
            repaint();
        }
    }
```

public void keyPressed(KeyEvent e) This method handles key press events. It handles debug mode, mic toggle, jump, fall, and pause keys.

```
    public void keyPressed(KeyEvent e) {
            // DEBUG
```

```
        if (DEBUGGER)
            if (e.getKeyChar() == '`') {
                debugMode = !debugMode;
            }

        // Mic
        if (e.getKeyChar() == 'm' || e.getKeyChar() == 'M') {
            toggleMic();
            mario.toggleMic();
            score.toggleMic();
        }

        // JUMP
        if (e.getKeyChar() == ' ' || e.getKeyChar() == 'w' || e.getKeyChar()
== 'W'
                || e.getKeyCode() == KeyEvent.VK_UP) {
            if (!paused && running) {
                mario.jump();
            } else if (paused && running) {
                resumeGame();
            }

            if (!running && !gameOver) {
                startGame();
                mario.run();
                mario.jump();
                introUI.overworld.stop();
            } else if (gameOver) {
                resetGame();
                introUI.overworld.stop();
            }
        }

        // FALL
        if (e.getKeyChar() == 's' || e.getKeyChar() == 'S' || e.getKeyCode()
== KeyEvent.VK_DOWN) {
            if (!paused && running) {
                mario.fall();
            }
        }

        // PAUSE
        if (e.getKeyChar() == 'p' || e.getKeyChar() == 'P' || e.getKeyCode()
== KeyEvent.VK_ESCAPE) {
            if (!paused && running) {
```

```
                pauseGame();
            } else if (paused && running) {
                resumeGame();
            }
        }
    }
```

Game Window

This class is responsible for setting up the main game window. It implements the GameSettings interface.

The **GameWindow()** constructor creates a new JFrame object, sets its size and resizability, adds a **GamePanel** (which presumably contains the game's visuals), adds a key listener to the frame, sets the default close operation, sets the window icon, positions the window, makes the window visible. It then starts the game loop.

```
public GameWindow() {
        JFrame mainGameWindow = new JFrame(WINDOW_TITLE); // Create the main game
window
        mainGameWindow.setSize(WINDOW_WIDTH, WINDOW_HEIGHT); // Set the window
size
        mainGameWindow.setResizable(WINDOW_RESIZABLE);

        GamePanel gamePanel = new GamePanel(); // Create the game panel
        mainGameWindow.add(gamePanel);

        mainGameWindow.addKeyListener(gamePanel); // Add the game panel as a key
listener

        mainGameWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Set the
window to exit on close (ensure the
                                                                    // program
closes when the window is closed)
        // Set the icon image for the JFrame
        ImageIcon icon = new ImageIcon("lib/icon_sm.png"); // app icon
        mainGameWindow.setIconImage(icon.getImage());
        // Set the window to the screen
        mainGameWindow.setLocationRelativeTo(null); // Set the window to the
center of the screen
        mainGameWindow.setVisible(true);
        mainGameWindow.setAlwaysOnTop(WINDOW_ALWAYS_ON_TOP); // Set the window
index (z)
```

```
        // Start the game loop
        startGameLoop(gamePanel);
    }
```

The **startGameLoop(GamePanel gamePanel)** method takes a **GamePanel** object as a parameter. Inside the method, a Timer object is created with a delay of **FRAME_TIME** milliseconds. The Timer is configured to call an ActionListener every **FRAME_TIME** milliseconds.

```
    /**
     * Starts the game loop for the given game panel.
     *
     * @param gamePanel the game panel to start the game loop for
     */
    private void startGameLoop(GamePanel gamePanel) {
        Timer timer = new Timer((int) FRAME_TIME, new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                // Update game logic
                gamePanel.validate();
                // Render the game
                gamePanel.repaint();
            }
        });
        timer.start();
    }
```

The ActionListener is an anonymous inner class that overrides the **actionPerformed** method. This method updates the game logic by calling **gamePanel.validate()** and then renders the game by calling **gamePanel.repaint()**.

Finally, the **start()** method is called on the Timer object to initiate the game loop.

Conclusion

In conclusion, "Super Mario Runner" represents a labor of love aimed at revitalizing the spirit of classic platformer games within the **Java Swing** framework, fusing the gameplay mechanics of **Chrome Dino** with the iconic character "Super Mario Bros".

Through meticulous design and development, this game pays homage to the nostalgia of yesteryear while offering modern accessibility and convenience. With its intuitive controls, vibrant graphics, and endless replayability, "**Super Mario Runner**" stands as a testament to the enduring appeal of retro gaming and the creativity of the Java programming language.