Space Invader

Group 7

Ms. Pema Yangden

Blockchain Development

Computer Science

Gyalpozhing College of Information Technology

20th November, 2023

Submitted by

Tandin Dorji (12210091)

Tashi Wangdi (12210095)

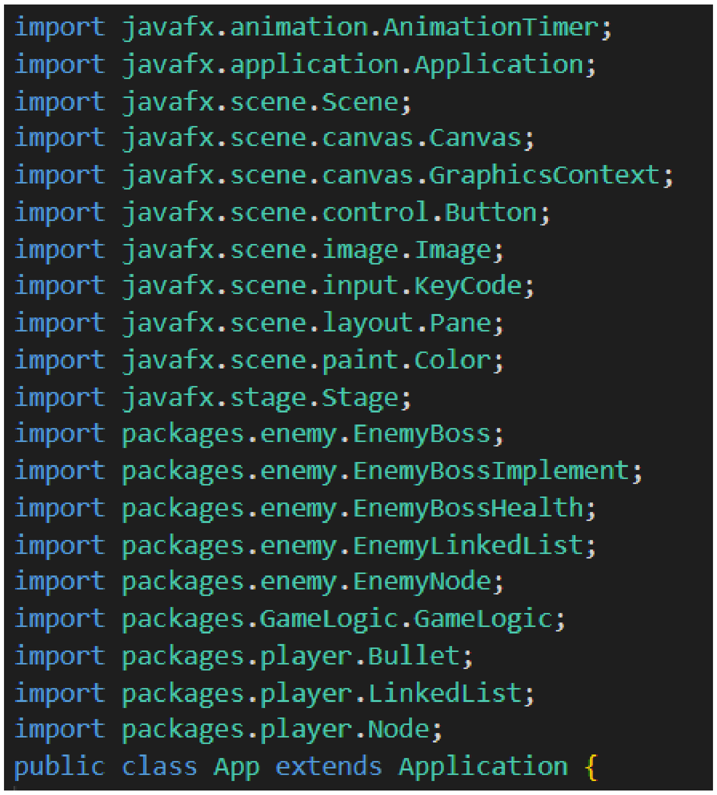
Yeshi Chogyel (12210102)

Jampel Dorji (12200047)

Documentation

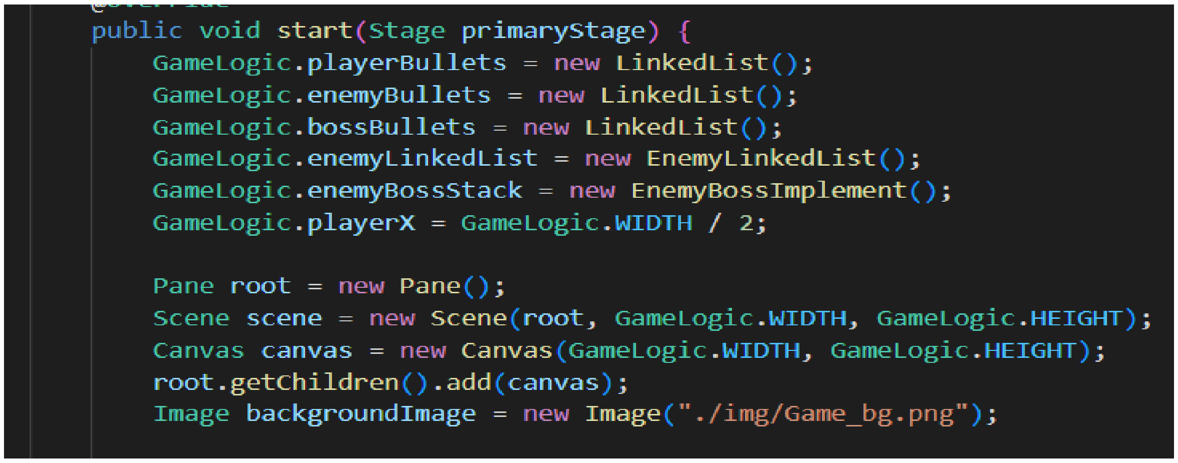
App.java

These are the java library packages used in app.java for the functionalities of the game.



```java
import javafx.animation.AnimationTimer;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.Button;
import javafx.scene.image.Image;
import javafx.scene.input.KeyCode;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import packages.enemy.EnemyBoss;
import packages.enemy.EnemyBossImplement;
import packages.enemy.EnemyBossHealth;
import packages.enemy.EnemyLinkedList;
import packages.enemy.EnemyNode;
import packages.GameLogic.GameLogic;
import packages.player.Bullet;
import packages.player.LinkedList;
import packages.player.Node;
public class App extends Application {
```

This is the start function where the game initializes essential components such as player bullets, enemy bullets, enemy ships and boss related structures. Initial player position and game dimensions are set.



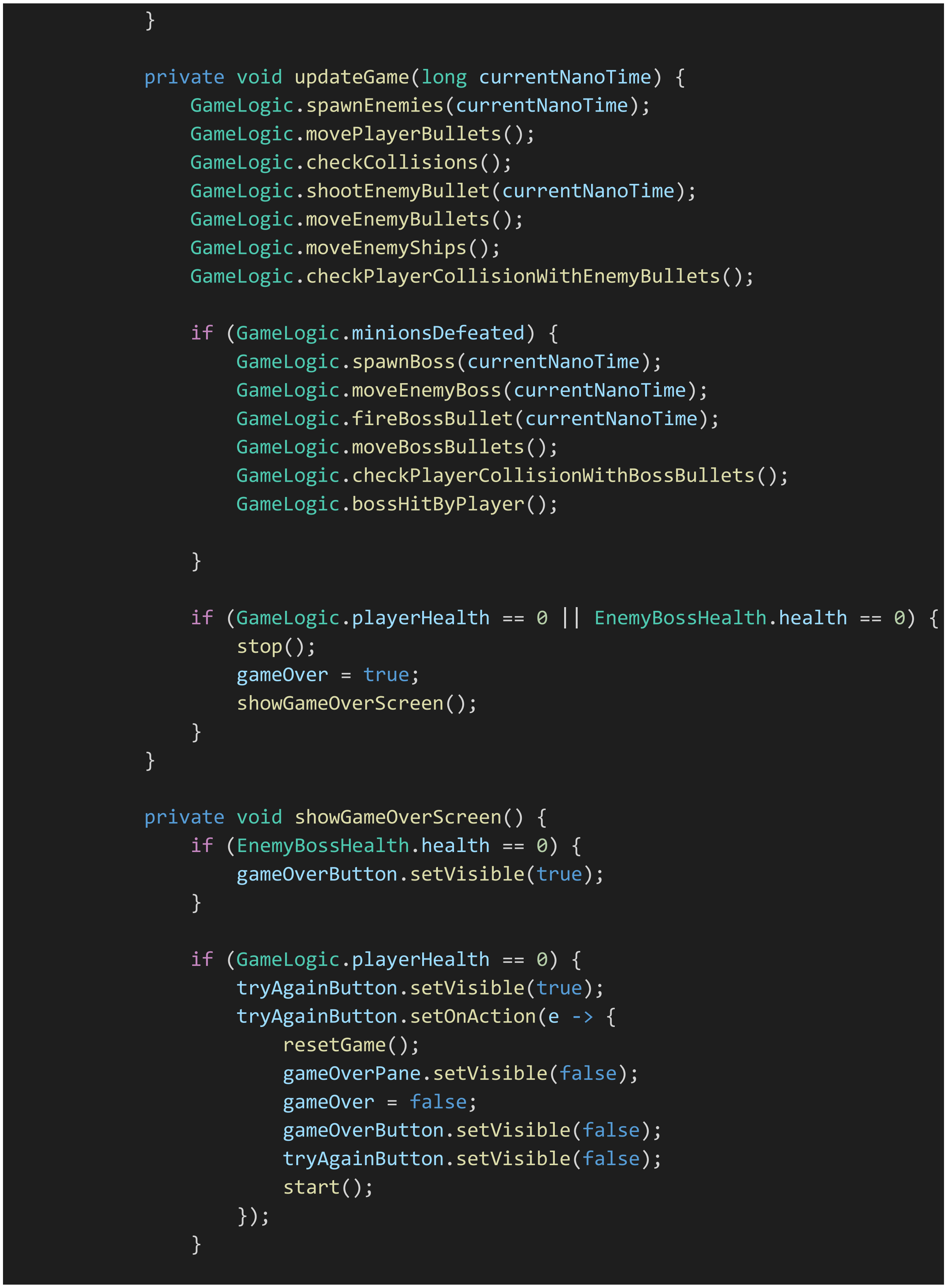
```java
    public void start(Stage primaryStage) {
        GameLogic.playerBullets = new LinkedList();
        GameLogic.enemyBullets = new LinkedList();
        GameLogic.bossBullets = new LinkedList();
        GameLogic.enemyLinkedList = new EnemyLinkedList();
        GameLogic.enemyBossStack = new EnemyBossImplement();
        GameLogic.playerX = GameLogic.WIDTH / 2;

        Pane root = new Pane();
        Scene scene = new Scene(root, GameLogic.WIDTH, GameLogic.HEIGHT);
        Canvas canvas = new Canvas(GameLogic.WIDTH, GameLogic.HEIGHT);
        root.getChildren().add(canvas);
        Image backgroundImage = new Image("./img/Game_bg.png");
```

In the code below, buttons for “Try Again” and “GameOver” are created but initially set to invisible and an overlay pane(“gameOverPane”) is created , styled with semi-transparent background.

```
        tryAgainButton = new Button("Try Again");
        tryAgainButton.setStyle("-fx-font-size: 20;");
        tryAgainButton.setTranslateX(GameLogic.WIDTH / 2 - 75);
        tryAgainButton.setTranslateY(GameLogic.HEIGHT / 2 - 25);
        tryAgainButton.setVisible(false);

        gameOverButton = new Button("Game Over");
        gameOverButton.setStyle("-fx-font-size: 20;");
        gameOverButton.setTranslateX(GameLogic.WIDTH / 2 - 75);
        gameOverButton.setTranslateY(GameLogic.HEIGHT / 2 - 25);
        gameOverButton.setVisible(false);

        Pane gameOverPane = new Pane();
        gameOverPane.setStyle("-fx-background-color: rgba(0, 0, 0, 0.7);");
        gameOverPane.getChildren().addAll(tryAgainButton, gameOverButton);
        root.getChildren().add(gameOverPane);

         // Set initial visibility of gameOverPane to false
        gameOverPane.setVisible(false);
```

In the code below, an animation Timer is used to create a game loop, the handle method continuously updates and renders the game graphics. The update Game method contains the core game logic, including enemy spawning, bullet movements, collision checks and boss related actions. There is also the logic for a player to play the game again after his health is equal to zero, we can reset the game using the reset function which is called when we click the “Try Again” button and the code calls the start function again.
There is also a function for “GameOverScreen” when either the player’s health or the Boss and minions health reaches zero. The gameOverPane is set visible and the player will be able to restart or reset the game.

```
 new AnimationTimer() {
            @Override
            public void handle(long currentNanoTime) {
                gc.drawImage(backgroundImage, 0, 0, GameLogic.WIDTH,
GameLogic.HEIGHT);
                updateGame(currentNanoTime);
                drawGame(gc);
```



```java
    }

    private void updateGame(long currentNanoTime) {
        GameLogic.spawnEnemies(currentNanoTime);
        GameLogic.movePlayerBullets();
        GameLogic.checkCollisions();
        GameLogic.shootEnemyBullet(currentNanoTime);
        GameLogic.moveEnemyBullets();
        GameLogic.moveEnemyShips();
        GameLogic.checkPlayerCollisionWithEnemyBullets();

        if (GameLogic.minionsDefeated) {
            GameLogic.spawnBoss(currentNanoTime);
            GameLogic.moveEnemyBoss(currentNanoTime);
            GameLogic.fireBossBullet(currentNanoTime);
            GameLogic.moveBossBullets();
            GameLogic.checkPlayerCollisionWithBossBullets();
            GameLogic.bossHitByPlayer();

        }

        if (GameLogic.playerHealth == 0 || EnemyBossHealth.health == 0) {
            stop();
            gameOver = true;
            showGameOverScreen();
        }
    }

    private void showGameOverScreen() {
        if (EnemyBossHealth.health == 0) {
            gameOverButton.setVisible(true);
        }

        if (GameLogic.playerHealth == 0) {
            tryAgainButton.setVisible(true);
            tryAgainButton.setOnAction(e -> {
                resetGame();
                gameOverPane.setVisible(false);
                gameOver = false;
                gameOverButton.setVisible(false);
                tryAgainButton.setVisible(false);
                start();
            });
        }
```

```
                gameOverPane.setVisible(true);
            }
        }.start();
```
In the code below, it handles the keyboard movements of a player to shoot and move the ship and also sets the title of the game.

```
scene.setOnKeyPressed(e -> {
            if (!gameOver) {
                KeyCode keyCode = e.getCode();
                if (keyCode == KeyCode.A) {
                    GameLogic.movePlayerLeft();
                } else if (keyCode == KeyCode.D) {
                    GameLogic.movePlayerRight();
                } else if (keyCode == KeyCode.W) {
                    GameLogic.fireBullet();
                }
            }
        });

        primaryStage.setScene(scene);
        primaryStage.setTitle("Galactic Invader");
        primaryStage.show();
    }
```

The drawGame function:

1. Draw Player: The player's spaceship is drawn using the drawImage method with the specified coordinates and dimensions.

2. Draw Player Bullets: Player bullets are drawn on the canvas using a loop that iterates through the linked list of player bullets. Bullets are represented as small white ovals (fillOval method).

3. Draw Enemy Bullets: Enemy bullets are drawn on the canvas using a loop that iterates through the linked list of enemy bullets. Bullets are represented as yellow characters (fillText method).

4. Draw Enemy Ships: Enemy ships are drawn on the canvas using a loop that iterates through the linked list of enemy ships. Enemy ships are represented as blue images (drawImage method).

5. Draw Boss and Bullets (if minions are defeated): If minions are defeated (GameLogic.minionsDefeated is true), the boss and its bullets are drawn on the canvas. Boss is represented as a green-yellow image, and bullets are represented as yellow characters.

6. Draw Boss Health Bar (if minions are defeated): If minions are defeated, the boss's health bar is drawn at the top left of the window. The health bar is a red rectangle whose width is proportional to the remaining boss health.

7. Draw Player Health Bar: The player's health bar is drawn at the top right of the window. The health bar is a green rectangle whose width is proportional to the remaining player health.

8. Draw Player Health Text: Text displaying the player's health is drawn next to the player's health bar.

```
 public void drawGame(GraphicsContext gc) {
        Image playerImage = new Image("./img/spaceship.png");
        double playerImageWidth = 25;
        double playerImageHeight = 25;
        double yOffset = 10;
        double playerHealthX = GameLogic.WIDTH - 150;
        double playerHealthY = 20;
        double playerHealthWidth = 100; // Width of the player health bar


        // Draw player
        gc.drawImage(playerImage, GameLogic.playerX, GameLogic.HEIGHT - 20 -
yOffset, playerImageWidth,
                playerImageHeight);

        // Draw player bullets
        Node current = GameLogic.playerBullets.head;
        while (current != null) {
            Bullet bullet = current.data;
            gc.setFill(Color.WHITE);
            gc.fillOval(bullet.x, bullet.y, 5, 5);
            current = current.next;
        }

        // Draw enemy bullets
        gc.setFill(Color.YELLOW);
        current = GameLogic.enemyBullets.head;
        while (current != null) {
            Bullet bullet = current.data;
            gc.fillText(String.valueOf(GameLogic.ENEMY_BULLET_CHAR), bullet.x,
bullet.y);
            current = current.next;
        }

        // Draw enemy ships
        Image enemyImage = new Image("./img/enemyShip.png");
        double enemyImageWidth = 25;
        double enemyImageHeight = 25;
        gc.setFill(Color.BLUE);
        EnemyNode enemy = GameLogic.enemyLinkedList.head;
        while (enemy != null) {
            gc.drawImage(enemyImage, enemy.x, enemy.y, enemyImageWidth,
enemyImageHeight);
            enemy = enemy.next;
        }

        if (GameLogic.minionsDefeated) {
            // Draw boss and its bullets
            Image bossImage = new Image("./img/boss_img.png");
            double bossImageWidth = 70;
            double bossImageHeight = 70;
            gc.setFill(Color.GREENYELLOW);
            EnemyBoss enemyboss = GameLogic.enemyBossStack.first;
            while (enemyboss != null) {
                gc.drawImage(bossImage, enemyboss.x, enemyboss.y, bossImageWidth,
bossImageHeight);
                enemyboss = enemyboss.next;
            }

            gc.setFill(Color.YELLOWGREEN);
            current = GameLogic.bossBullets.head;
            while (current != null) {
                Bullet bullet = current.data;
                gc.fillText(String.valueOf(GameLogic.ENEMY_BULLET_CHAR),
bullet.x, bullet.y);
                current = current.next;
            }

            // Draw boss health bar at the top left of the window
            double bossHealthX = 20; // Adjusted X-coordinate for boss health bar
            double bossHealthY = 20; // Adjusted Y-coordinate for boss health bar
            double bossHealthWidth = 150; // Width of the boss health bar
            double bossHealthHeight = 10; // Height of the boss health bar
            gc.setFill(Color.RED);
            gc.fillRect(bossHealthX, bossHealthY, bossHealthWidth *
(EnemyBossHealth.health / 10.0), bossHealthHeight);
            gc.setFill(Color.WHITE);
            gc.fillText("Boss Health:", bossHealthWidth, bossHealthHeight + 5);
        }

        // Draw player health bar
        double playerHealthHeight = 10; // Height of the player health bar
        gc.setFill(Color.GREEN);
        gc.fillRect(playerHealthX, playerHealthY, playerHealthWidth *
(GameLogic.playerHealth / 10.0),
                playerHealthHeight);

        // Draw player health text
        gc.setFill(Color.WHITE);
        gc.fillText("Player Health: " + GameLogic.playerHealth, playerHealthX,
playerHealthY - 5);

    }
```

The next code are for Enemy functionalities which are in Enemy package.

EnemyBoss.java

The below code defines a class called EnemyBoss in the packages.enemy package. This class represents an individual enemy boss in a game and contains public attributes for its position (x and y). The class has a constructor that initializes the boss with specific coordinates, and the next reference is set to null by default. This allows for the creation of linked structures of enemy boss, providing a flexible and organized way to manage the enemy boss in the game.

```
package packages.enemy;

public class EnemyBoss {
    public EnemyBoss next;
    public int x;
    public int y;

    public EnemyBoss(int x, int y) {
        this.x = x;
        this.y = y;
        next = null;
    }
}
```

EnemyBossHealth.java

The below code defines a class called EnemyBossHealth in the packages.enemy package. It sets the boss health.

```
package packages.enemy;

public class EnemyBossHealth {
    public static int health = 20;
}
```

EnemyBossImplement.java

The below code defines a class called EnemyBossImplement in the packages.enemy. The EnemyBossImplement class uses basic stack implementation to manage instances of the EnemyBoss class. It allows pushing enemy boss onto the stack, popping it off and checking if the stack is empty.

```
package packages.enemy;

public class EnemyBossImplement {

    public static EnemyBoss first;
    public static int size;

    public EnemyBossImplement() {
        first = null;
        size = 0;
    }

    public void push(int x, int y) {
        EnemyBoss newNode = new EnemyBoss(x, y);
        newNode.next = first;
        first = newNode;
        size++;
    }

    public void pop() {
        first = first.next;
        size--;
    }

    public boolean isStackEmpty() {
        return size == 0;
    }

    public void clear() {
    }

}
```

EnemyLinkedList.java

The EnemyLinkedList class uses a linked list implementation for managing instances of the EnemyNode class. It supports adding new enemy nodes to the end of the list and deleting specified enemy nodes.

```
package packages.enemy;

public class EnemyLinkedList {
    public EnemyNode head;
    public EnemyNode tail;

    public EnemyLinkedList() {
        head = null;
        tail = null;
    }

    public void insertEnemy(int x, int y) {
        EnemyNode newNode = new EnemyNode(x, y);
        if (head == null) {
            head = newNode;
            tail = newNode;
        } else {
            tail.next = newNode;
            tail = newNode;
        }
    }

    public void deleteEnemy(EnemyNode enemy) {
        if (head == enemy) {
            head = head.next;
            if (head == null) {
                tail = null;
            }
            return;
        }

        EnemyNode current = head;
        while (current != null && current.next != enemy) {
            current = current.next;
        }
        if (current != null) {
            current.next = enemy.next;
            if (enemy == tail) {
                tail = current;
            }
        }
    }

    public void clear() {
    }
}
```

EnemyNode.java

The EnemyNode class uses a linked list for constructing enemy ships in a game. Each EnemyNode instance represents an individual enemy ship, storing its x and y coordinates. The next attribute allows the creation of a linked structure, enabling the organization and manipulation of enemy ships in a sequential manner.

```
package packages.enemy;

public class EnemyNode {
    public int x;
    public int y;
    public EnemyNode next;

    public EnemyNode(int x, int y) {
        this.x = x;
        this.y = y;
        this.next = null;
    }
}
```

The next code are for player functionalities which are in player package.

Bullet.java

The below code uses a class named Bullet within the packages.player package. This class represents a bullet entity in a game, and each instance of this class contains information about the bullet's position (x and y coordinates) and characteristics (whether it is a player bullet or a boss bullet).

```
package packages.player;

public class Bullet {
    public int x;
    public int y;
    boolean isPlayerBullet;
    boolean isBossBullet;

    public Bullet(int x, int y, boolean isPlayerBullet, boolean isBossBullet) {
        this.x = x;
        this.y = y;
        this.isPlayerBullet = isPlayerBullet;
        this.isBossBullet = isBossBullet;
    }
}
```

LinkedList.java

The LinkedList class uses a linked list implementation to manage instances of the Bullet class. It includes methods to retrieve the size of the list, insert new bullets, and delete specific bullets. This linked list is used for organizing and manipulating bullet instances in a game.

```java
package packages.player;

public class LinkedList {
    public Node head, tail;

    public LinkedList() {
        head = null;
        tail = null;
    }

    public int size() {
        Node current = head;
        int count = 0;
        while (current != null) {
            count++;
            current = current.next;
        }
        return count;
    }

    public void insertBullet(int x, int y, boolean isPlayerBullet, boolean
isBossBullet) {
        Bullet bullet = new Bullet(x, y, isPlayerBullet, isBossBullet);
        Node newNode = new Node(bullet);
        if (head == null) {
            head = newNode;
            tail = newNode;
        } else {
            tail.next = newNode;
            tail = newNode;
        }
    }

    public void deleteBullet(Bullet bullet) {
        if (head == null) {
            return;
        }
        if (head.data == bullet) {
            head = head.next;
            if (head == null) {
                tail = null;
            }
            return;
        }
        Node current = head;
        while (current.next != null) {
            if (current.next.data == bullet) {
                current.next = current.next.next;
                if (current.next == null) {
                    tail = current;
                }
                return;
            }
            current = current.next;
        }
    }

}
```

Node.java

The Node class uses a linked list for constructing Bullet objects in a game. Each instance of this class represents a node in the linked list, storing a reference to a Bullet and providing a reference to the next node. This class facilitates the creation of a sequential data structure, allowing for the organization and manipulation of bullets within the context of a game. It is a key component in the implementation of the linked list used to manage player bullets.

```java
package packages.player;

public class Node {
    public Bullet data;
    public Node next;

    public Node(Bullet data) {
        this.data = data;
        next = null;
    }
}
```

The next code are for Game Logic Functions which are in Game Logic package.

GameLogic.java

The Game Logic Variables for the code given below:

1. Constants:

WIDTH and HEIGHT: Define the dimensions of the game window.
PLAYER_CHAR, PLAYER_BULLET_CHAR, ENEMY_CHAR, and ENEMY_BULLET_CHAR: Characters representing the player, player's bullets, enemies, and enemy bullets, respectively.

BULLET_SPEED: Speed of bullets.

ENEMY_FIRE_RATE: Rate at which enemies fire.

ENEMY_BULLET_RATE: Rate at which enemy bullets are fired (currently set to 0).

ENEMY_BULLET_INTERVAL and BOSS_BULLET_INTERVAL: Time intervals between enemy and boss bullet spawns.

ENEMY_MOVE_INTERVAL: Time interval for enemy ship movement.

PLAYER_SPEED: Speed of the player's movement.

2. Game State Variables:

isGameOver: Indicates whether the game is over.

enemySpawnedCount and maxEnemies: Count of spawned enemies and the maximum allowed enemies.

lastEnemyFireTime, lastEnemyMoveTime, lastBossMoveTime, and lastBossBulletTime: Store the time of the last corresponding events.

playerX: X-coordinate of the player.

playerHealth: Player's health.

minionsDefeated: Indicates whether all regular enemies are defeated.

startX and startY: Initial coordinates for spawning enemies.

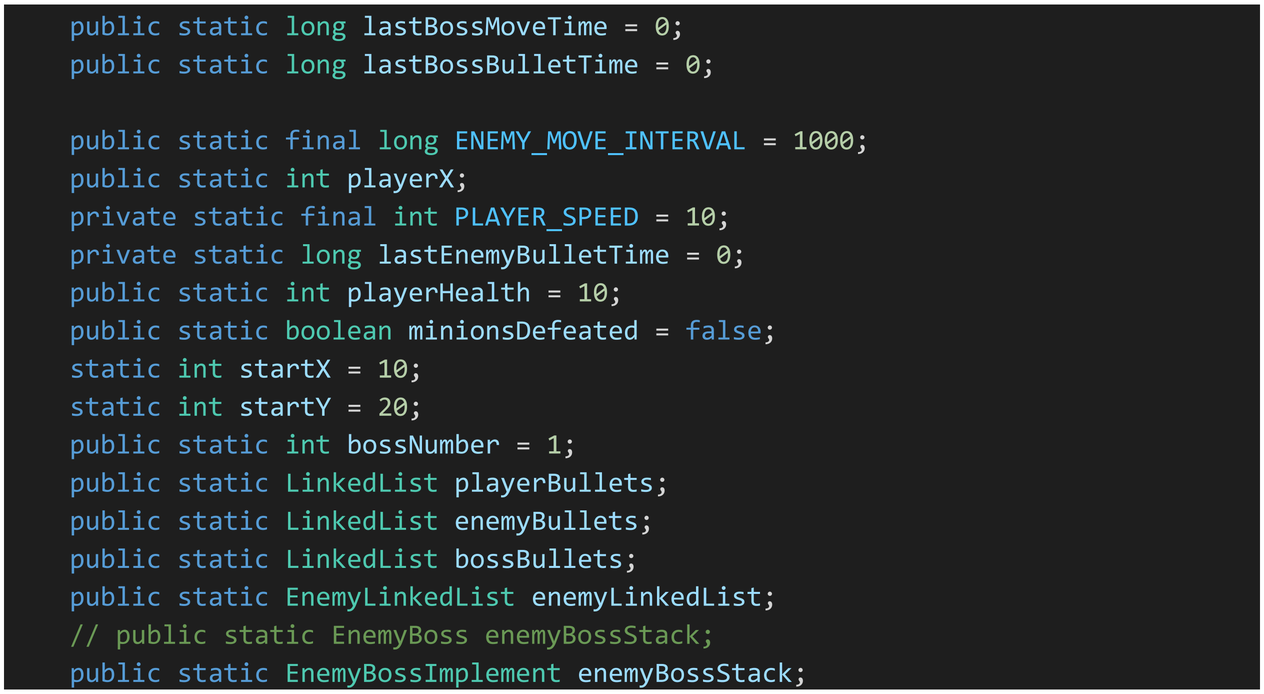
bossNumber: Number of bosses to be spawned.

playerBullets, enemyBullets, and bossBullets: Linked lists storing player, enemy, and boss bullets.

enemyLinkedList: Linked list storing enemy ships.

enemyBossStack: Implementation for managing boss entities.

```java
public static final int WIDTH = 600;
public static final int HEIGHT = 600;
public static final char PLAYER_CHAR = '^';
public static final char PLAYER_BULLET_CHAR = 'o';
public static final char ENEMY_CHAR = '8';
public static final char ENEMY_BULLET_CHAR = '+';
public static final int BULLET_SPEED = 3;
public static final int ENEMY_FIRE_RATE = 40;
public static final int ENEMY_BULLET_RATE = 0;
private static final int ENEMY_BULLET_INTERVAL = 1000;
private static final int BOSS_BULLET_INTERVAL = 1000;

public static boolean isGameOver;
public static int enemySpawnedCount = 0;
public static int maxEnemies = 5;
public static long lastEnemyFireTime = 0;
public static long lastEnemyMoveTime = 0;
```



```java
    public static long lastBossMoveTime = 0;
    public static long lastBossBulletTime = 0;

    public static final long ENEMY_MOVE_INTERVAL = 1000;
    public static int playerX;
    private static final int PLAYER_SPEED = 10;
    private static long lastEnemyBulletTime = 0;
    public static int playerHealth = 10;
    public static boolean minionsDefeated = false;
    static int startX = 10;
    static int startY = 20;
    public static int bossNumber = 1;
    public static LinkedList playerBullets;
    public static LinkedList enemyBullets;
    public static LinkedList bossBullets;
    public static EnemyLinkedList enemyLinkedList;
    // public static EnemyBoss enemyBossStack;
    public static EnemyBossImplement enemyBossStack;
```

The code below functions:

1. **Spawn Boss (spawnBoss):**

This function checks if the game is not over, all regular enemies are defeated (minionsDefeated), and the number of bosses spawned (EnemyBossImplement.size) is less than the specified bossNumber. If these conditions are met, a new boss entity is pushed onto the boss stack.

2. **Fire Boss Bullet (fireBossBullet):**

Fires bullets from each existing boss entity at a regular interval. It iterates through the boss entities, retrieves their positions, and inserts bullets into the bossBullets linked list.

3. **Move Boss Bullets (moveBossBullets):**

Moves boss bullets upward on the screen. If a bullet goes beyond the game window (HEIGHT), it is deleted from the bossBullets linked list.

4. **Move Enemy Boss (moveEnemyBoss):**

Moves boss entities horizontally, alternating between left and right movements. It uses a random direction (moveDirection) and ensures that the boss entities stay within the game window.

5. **Check Player Collision with Boss Bullets (checkPlayerCollisionWithBossBullets):**

Iterates through boss bullets, checks if the player is hit by any of them, and updates the game state accordingly.

```java
 public static void spawnBoss(long currentNanoTime) {
        if (!isGameOver && minionsDefeated && EnemyBossImplement.size <
bossNumber) {
            int centerX = WIDTH / 2;
            enemyBossStack.push(centerX, startY + 20);
            // bossBullets.insertBullet(centerX, startY + 20, false, true);
        }
    }

    public static void fireBossBullet(long currentNanoTime) {
        if (currentNanoTime - lastBossBulletTime > BOSS_BULLET_INTERVAL *
1_000_000) {
            EnemyBoss boss = enemyBossStack.first;
            while (boss != null) {
                int bulletX = boss.x;
                int bulletY = boss.y;
                bossBullets.insertBullet(bulletX, bulletY, false, true);
                boss = boss.next;
            }
            lastBossBulletTime = currentNanoTime;
        }
    }

    public static void moveBossBullets() {
        Node current = bossBullets.head;

        while (current != null) {
            Bullet bullet = current.data;
            int bulletY = bullet.y + BULLET_SPEED;
            if (bulletY > HEIGHT) {
                bossBullets.deleteBullet(bullet);
            } else {
                bullet.y = bulletY;
            }

            current = current.next;
        }
    }

    public static void moveEnemyBoss(long currentNanoTime) {
        long currentTime = System.currentTimeMillis();

        if (currentTime - lastBossMoveTime >= 1000) {
            int moveDirection = Math.random() < 0.5 ? -1 : 1;

            EnemyBoss boss = enemyBossStack.first;
            while (boss != null) {
                int newX = boss.x + moveDirection * 10;

                newX = Math.max(10, Math.min(WIDTH - 10, newX));

                boss.x = newX;

                boss = boss.next;
            }

            lastBossMoveTime = currentTime;
        }
    }

    public static void checkPlayerCollisionWithBossBullets() {
        Node bulletNode = bossBullets.head;
        while (bulletNode != null) {
            Bullet bossBullet = bulletNode.data;

            if (playerHitByBoss(bossBullet, playerX, HEIGHT - 20)) {
                bossBullets.deleteBullet(bossBullet);
                playerIsHit();

            }

            bulletNode = bulletNode.next;
        }
    }
```

6. **Player Hit (playerHit):**

This function determines if the player is hit by an enemy bullet. It calculates the Euclidean distance between the player's position and the bullet's position. If the distance is less than a predefined threshold (collisionThreshold), it returns true, indicating a collision.

7. **Player Hit By Boss (playerHitByBoss):**

Similar to the playerHit function, this one checks if the player is hit by a boss bullet. It calculates the Euclidean distance between the player's position and the boss bullet's position. If the distance is less than the collisionThreshold, it returns true, indicating a collision.

```
    public static void playerIsHit() {
        playerHealth--;
        System.out.println("You have been hit! Your remaining health is " +
playerHealth);

        if (playerHealth <= 0) {
            isGameOver = true;
        }
    }

    public static boolean playerHit(Bullet bullet, int playerX, int playerY) {
        double collisionThreshold = 10.0;

        double dx = bullet.x - playerX;
        double dy = bullet.y - playerY;
        double distance = Math.sqrt(dx * dx + dy * dy);

        return distance < collisionThreshold;
    }
```

8. **Spawn Enemies (spawnEnemies):**

This function spawns regular enemies at predefined intervals if the game is not over, the maximum number of enemies (maxEnemies) hasn't been reached, and a specified time has passed since the last enemy spawn. It inserts the enemy into the enemyLinkedList and fires a bullet from the spawned enemy.

9. **Move Enemy Ships (moveEnemyShips):**

Moves regular enemy ships horizontally, changing rows when reaching the right edge of the screen. It ensures a time interval (ENEMY_MOVE_INTERVAL) between movements.

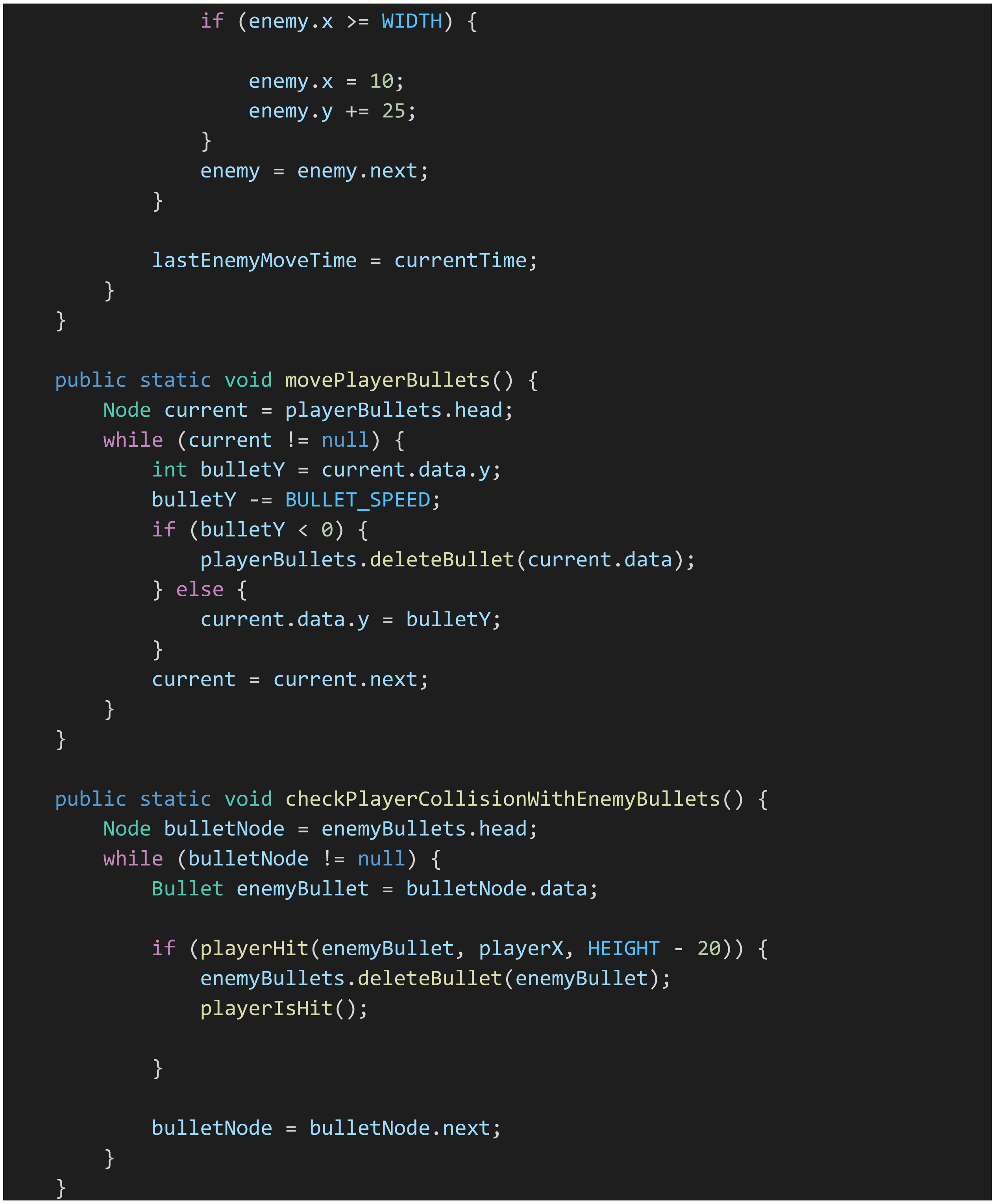
10. **Move Player Bullets (movePlayerBullets):**

Moves player bullets upward on the screen and deletes them if they go beyond the game window.

11. **Check Player Collision with Enemy Bullets (checkPlayerCollisionWithEnemyBullets):**

Iterates through enemy bullets, checks if the player is hit by any of them, and updates the game state accordingly.

```
 public static void spawnEnemies(long currentNanoTime) {
        if (!isGameOver && enemySpawnedCount < maxEnemies
                && currentNanoTime - lastEnemyFireTime > ENEMY_FIRE_RATE *
1_000_000) {
            enemyLinkedList.insertEnemy(startX, startY);
            enemyBullets.insertBullet(startX, startY, false, false);
            lastEnemyFireTime = currentNanoTime;
            enemySpawnedCount++;
            startX += 25;
            if (startX >= WIDTH) {
                startX = 10;
                startY += 25;
            }
        }
    }

    public static void moveEnemyShips() {
        long currentTime = System.currentTimeMillis();

        if (currentTime - lastEnemyMoveTime >= ENEMY_MOVE_INTERVAL) {
            EnemyNode enemy = enemyLinkedList.head;
            while (enemy != null) {
                enemy.x += 10;
```



```java
                if (enemy.x >= WIDTH) {

                    enemy.x = 10;
                    enemy.y += 25;
                }
                enemy = enemy.next;
            }

            lastEnemyMoveTime = currentTime;
        }
    }

    public static void movePlayerBullets() {
        Node current = playerBullets.head;
        while (current != null) {
            int bulletY = current.data.y;
            bulletY -= BULLET_SPEED;
            if (bulletY < 0) {
                playerBullets.deleteBullet(current.data);
            } else {
                current.data.y = bulletY;
            }
            current = current.next;
        }
    }

    public static void checkPlayerCollisionWithEnemyBullets() {
        Node bulletNode = enemyBullets.head;
        while (bulletNode != null) {
            Bullet enemyBullet = bulletNode.data;

            if (playerHit(enemyBullet, playerX, HEIGHT - 20)) {
                enemyBullets.deleteBullet(enemyBullet);
                playerIsHit();

            }

            bulletNode = bulletNode.next;
        }
    }
```

12. **Move Enemy Bullets (moveEnemyBullets):**

Moves enemy bullets downward on the screen and deletes them if they go beyond the game window.

13. **Shoot Enemy Bullet (shootEnemyBullet):**

Shoots bullets from regular enemies at a regular interval. It iterates through regular enemies, retrieves their positions, and inserts bullets into the enemyBullets linked list.

14. **Check Collisions (checkCollisions):**

Iterates through player bullets, checks collisions with regular enemies and boss entities, and updates the game state accordingly. If all regular enemies are defeated, it sets minionsDefeated to true.

15. **Boss Hit By Player (bossHitByPlayer):**

Iterates through player bullets, checks collisions with boss entities, and updates the boss's health. If a collision is detected, the player bullet is deleted.

```java
    public static void moveEnemyBullets() {
        Node current = enemyBullets.head;
        while (current != null) {
            int bulletY = current.data.y;
            bulletY += BULLET_SPEED;
            if (bulletY > HEIGHT) {
                enemyBullets.deleteBullet(current.data);
            } else {
                current.data.y = bulletY;
            }
            current = current.next;
        }
    }

    public static void shootEnemyBullet(long currentNanoTime) {
        if (currentNanoTime - lastEnemyBulletTime > ENEMY_BULLET_INTERVAL *
1_000_000) {
            EnemyNode enemy = enemyLinkedList.head;
            while (enemy != null) {
                int bulletX = enemy.x;
                int bulletY = enemy.y;
                enemyBullets.insertBullet(bulletX, bulletY, false, false);
                enemy = enemy.next;
            }
            lastEnemyBulletTime = currentNanoTime;
        }
    }
```

```
public static void bossHitByPlayer() {
        Node playerBulletNode = playerBullets.head;
        while (playerBulletNode != null) {
            Bullet playerBullet = playerBulletNode.data;

            Node currentBulletNode = playerBulletNode;

            EnemyBoss enemyBossNode = enemyBossStack.first;
            if (isCollisionWithBoss(playerBullet, enemyBossNode)) {
                playerBullets.deleteBullet(playerBullet);
                EnemyBossHealth.health--;
                System.out.println(EnemyBossHealth.health);
                break;
            }
            // while (enemyBossNode != null) {
            // EnemyBoss boss = enemyBossNode;

            // boss = boss.next;
            // }

            playerBulletNode = currentBulletNode.next;
        }
    }

    public static boolean isCollisionWithBoss(Bullet bullet, EnemyBoss boss) {
        double collisionThreshold = 10.0;

        double dx = bullet.x - boss.x;
        double dy = bullet.y - boss.y;
        double distance = Math.sqrt(dx * dx + dy * dy);

        return distance < collisionThreshold;
    }

    public static void checkCollisions() {
        Node playerBulletNode = playerBullets.head;
        while (playerBulletNode != null) {
            Bullet playerBullet = playerBulletNode.data;

            Node currentBulletNode = playerBulletNode;

            EnemyNode enemyNode = enemyLinkedList.head;
            while (enemyNode != null) {
                EnemyNode enemy = enemyNode;

                if (isCollision(playerBullet, enemy)) {

                    playerBullets.deleteBullet(playerBullet);
                    enemyLinkedList.deleteEnemy(enemy);
                    break;
                }

                enemyNode = enemyNode.next;
            }

            playerBulletNode = currentBulletNode.next;
        }
        if (enemyLinkedList.head == null) {
            minionsDefeated = true;
        }
    }
```