PvZ-deeptry

Members

Lý Khải Minh - ITITSB22008

Đỗ Minh Duy - ITITSB22029

Table of Contents

I. INTRODUCTION

The "Plant vs Zombie" project is a simplified version of the popular tower defense game where players place plants to defend against waves of zombies. This project leverages Object-Oriented Programming (OOP) principles to create a modular, maintainable, and extensible codebase. The game is designed with several key classes representing different elements of the game, such as plants, zombies, projectiles, and the game board.

The Random class in your code provides methods to generate random integer and double values within a specified range.

1. About Plant vs Zombies

Plants vs. Zombies (abbreviated as PvZ or PvZ1) is a tower defense video game developed and originally published by PopCap Games and it is the first installment in the Plants vs. Zombies series. The game involves homeowner who uses a variety of different plants to prevent waves of zombies from entering his house and "eating his brain".

2. About the game project

In our game, we implemented a simplified version of the "Plants vs. Zombie" game by applying OOP principles such as encapsulation, inheritance, and polymorphism to ensure the code is modular and maintainable. Besides, we also create an engaging and interactive game experience.

II. SOFTWARE REQUIREMENTS

1. Tools and platforms

a. UML making (Visualize the game)

b. Visual Studio Code (IDE)

c. GitHub Desktop (Commits, merge branches)

d. Photoshop (Editing images, background)

e. Greenfoot (Framework)

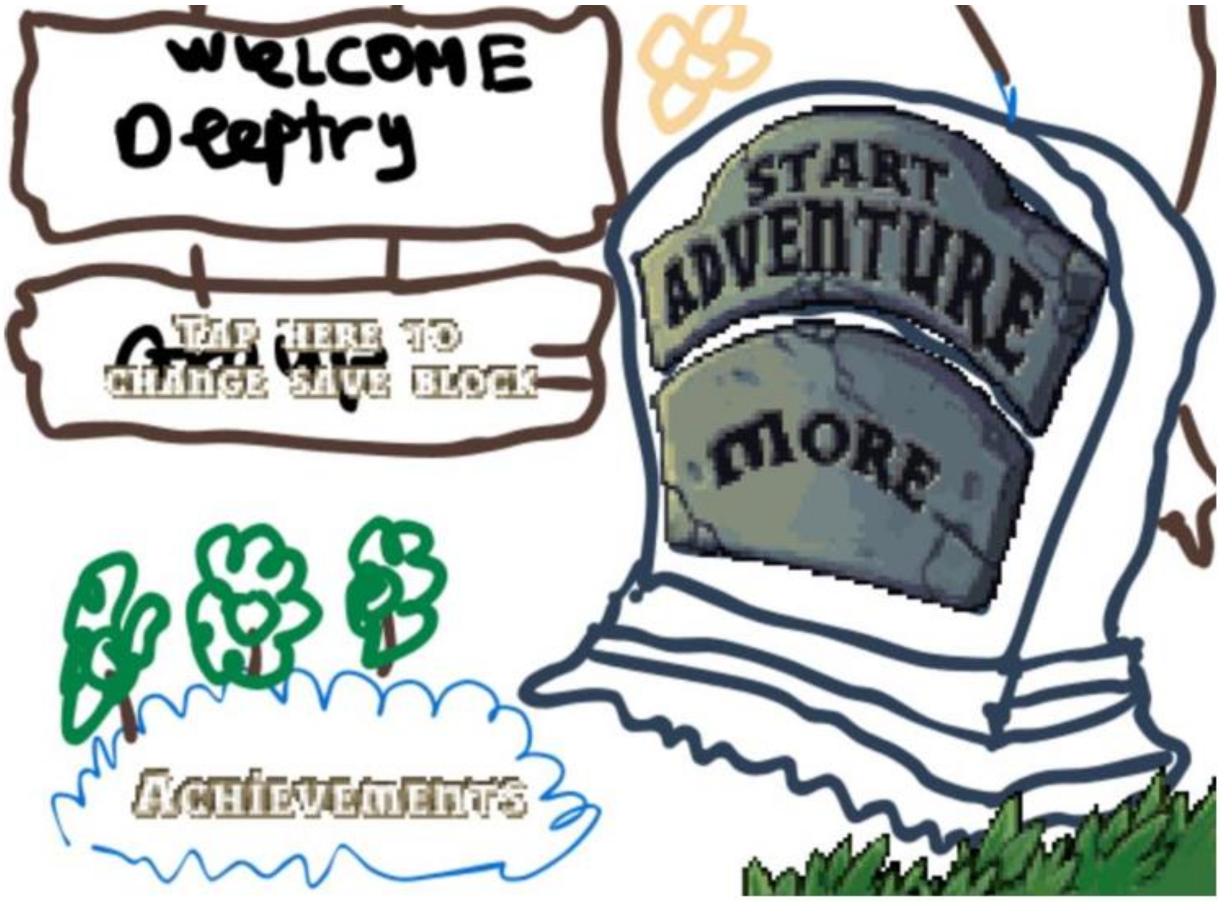
2. About Greenfoot:

Greenfoot is an educational software designed to teach programming concepts through the creation of interactive graphical applications, such as simulations and games. Developed by the King's College London, it leverages the Java programming language in an engaging and user-friendly environment.

Greenfoot provides a visual and interactive platform where users can write code to control actors in a 2D world, making it an ideal tool for beginners and educators to introduce Object-oriented Programming. Its intuitive interface and extensive documentation support a hands-on learning approach, allowing users to see the immediate effects of their code, thereby reinforcing programming concepts and logic in a practical, enjoyable manner.

III. GAME RULES

1. Main menu

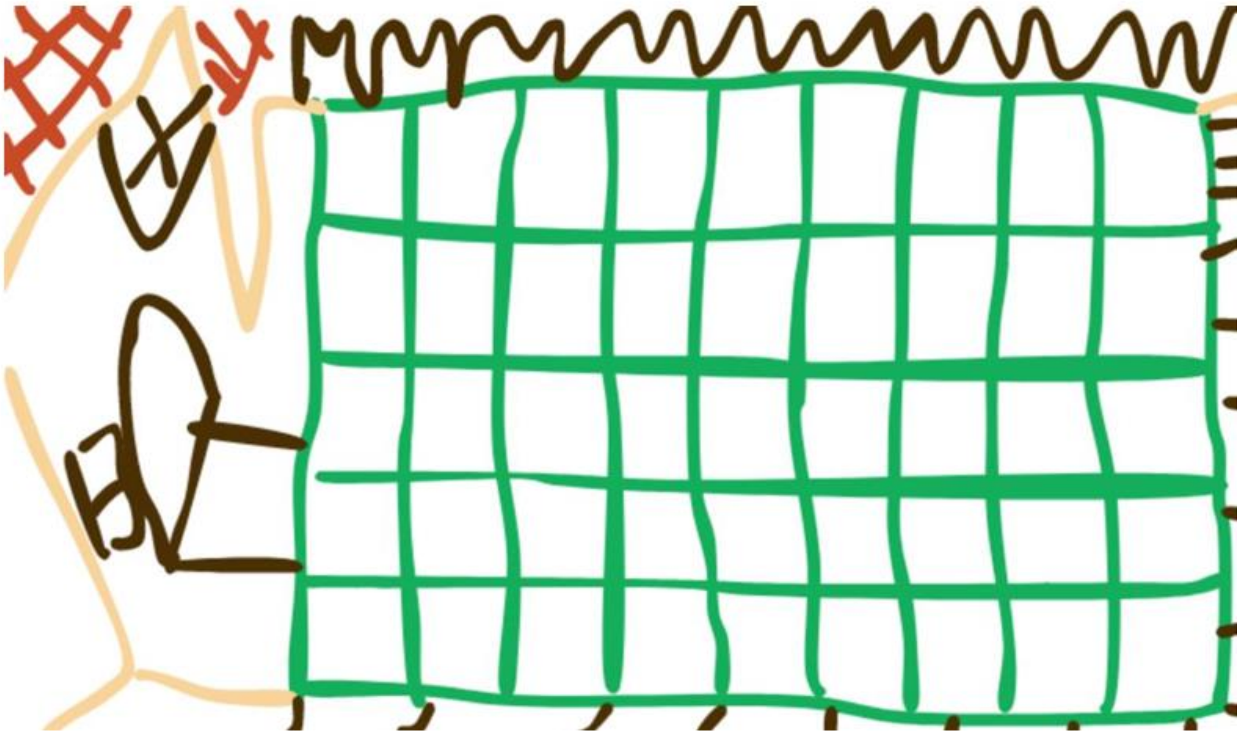


First start at the menu

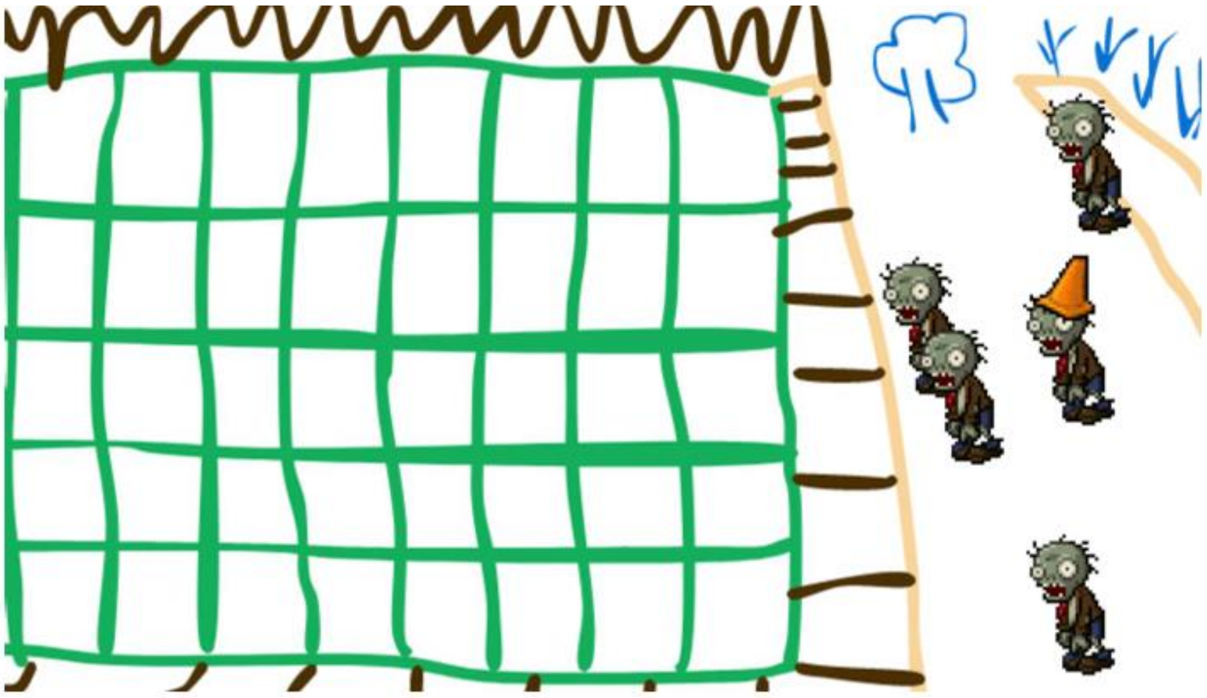
Click on "**START ADVENTURE**" to play the game

2. Gameplay

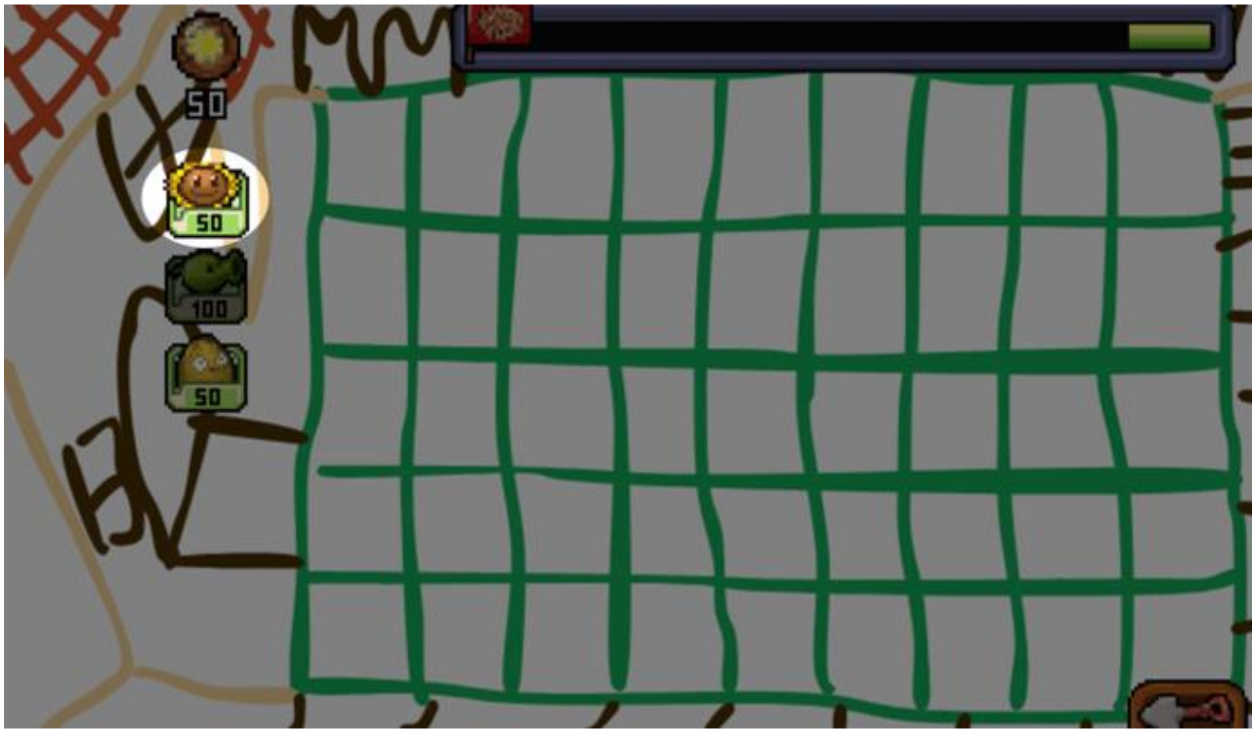
You will see the big, long frontyard of your house.



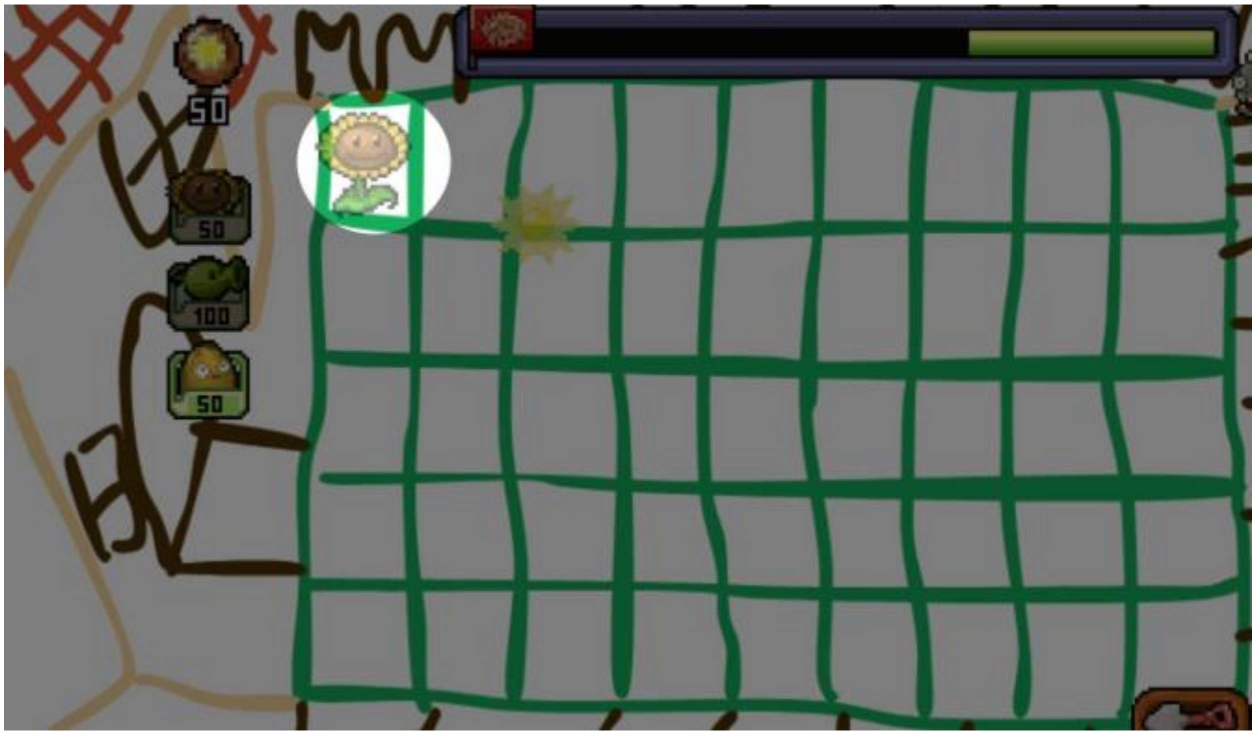
And the game will show you the types of zombies appear in that level.



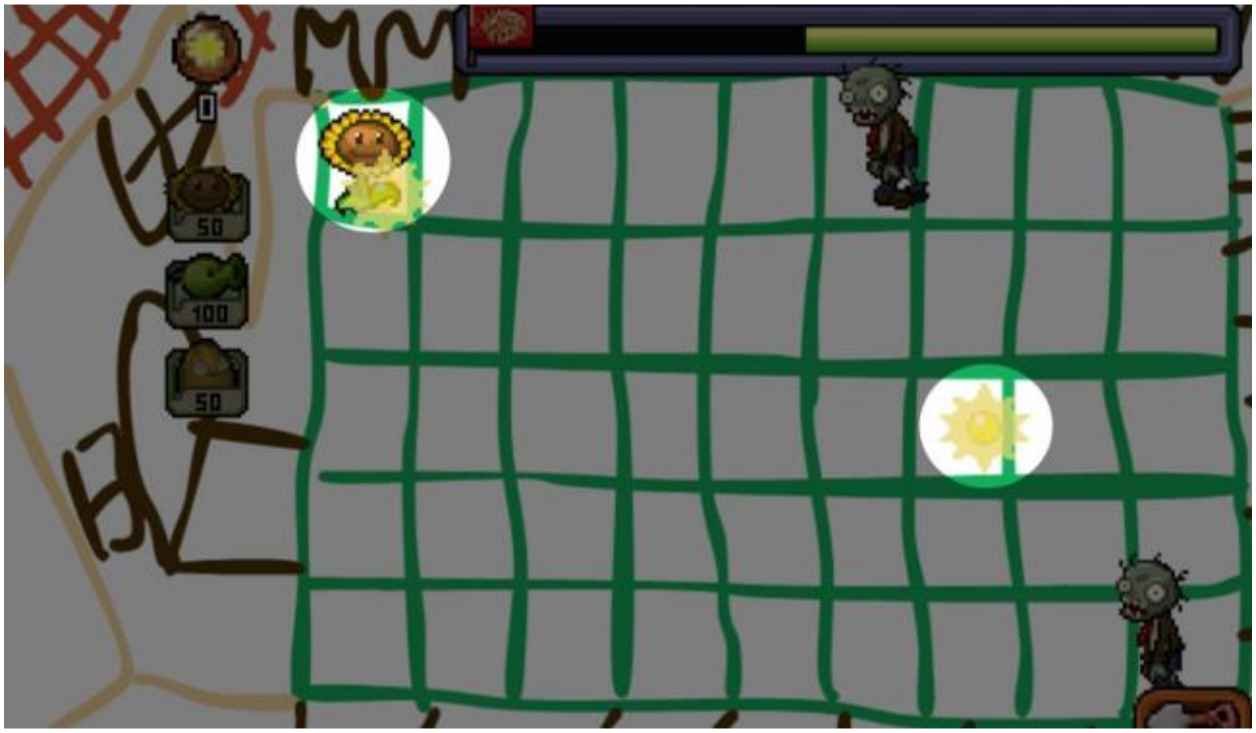
First select the SunFlower



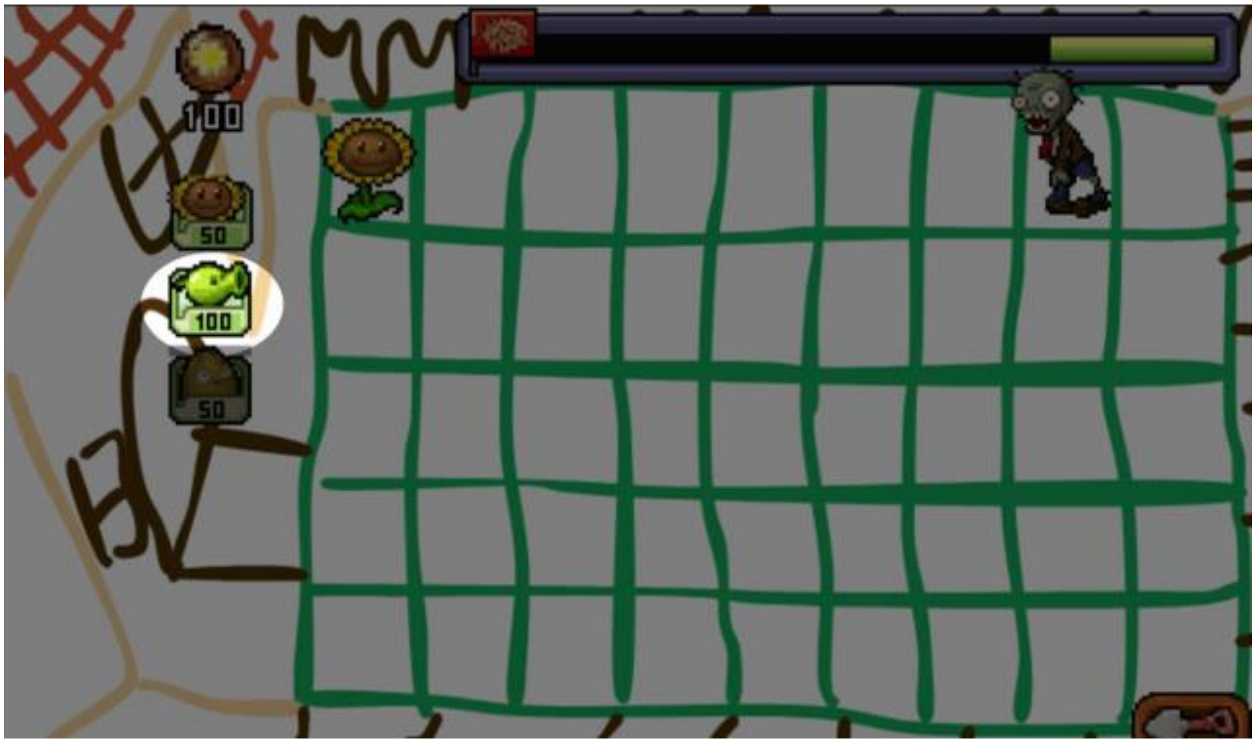
Then put it in the top left square.



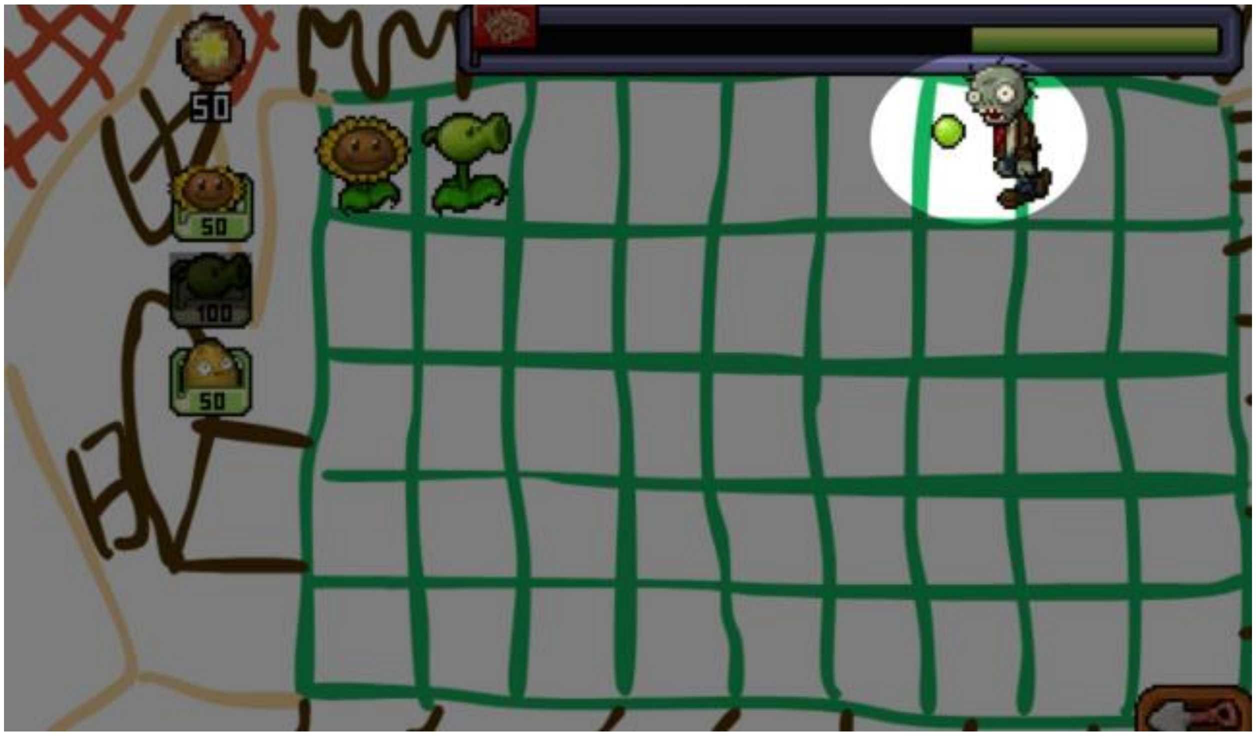
Collect sun from flower and The Sun.



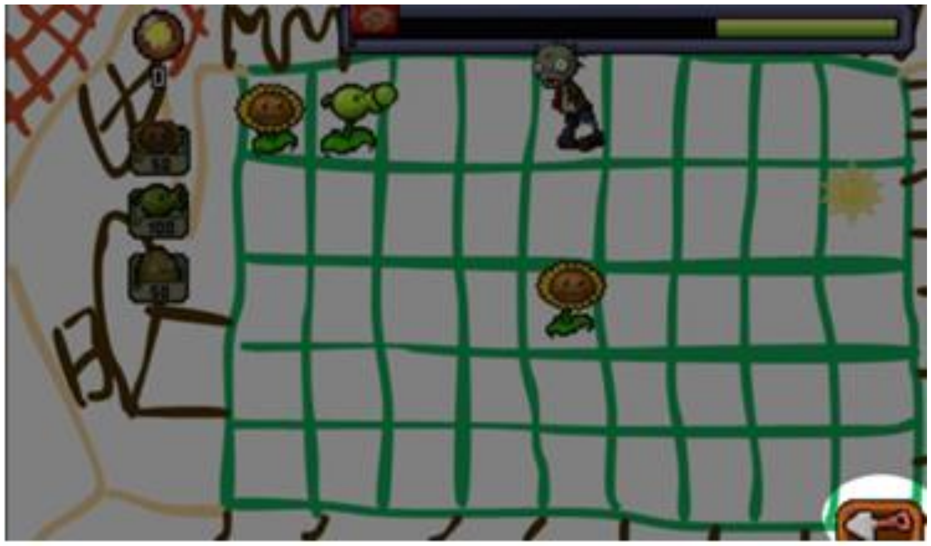
When you have 100 sun, you can buy PeaShooter to atack the zombies.

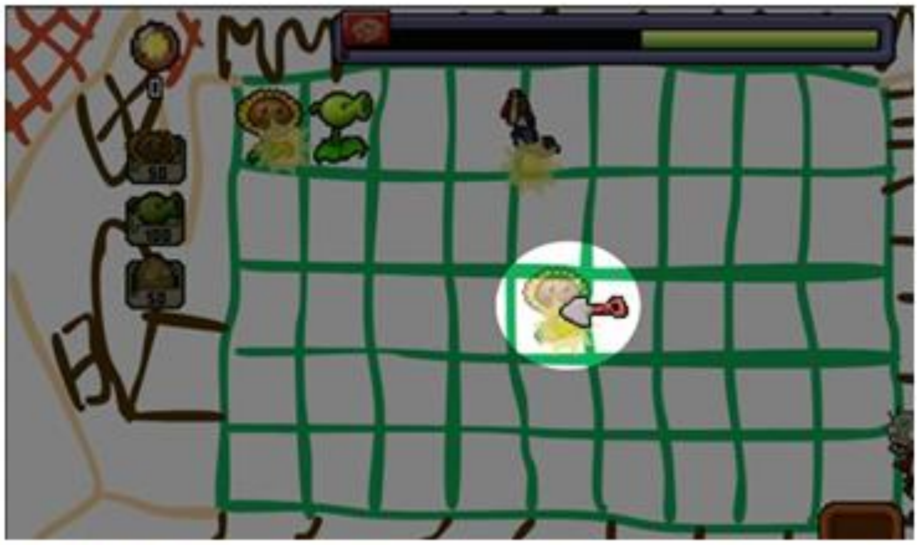


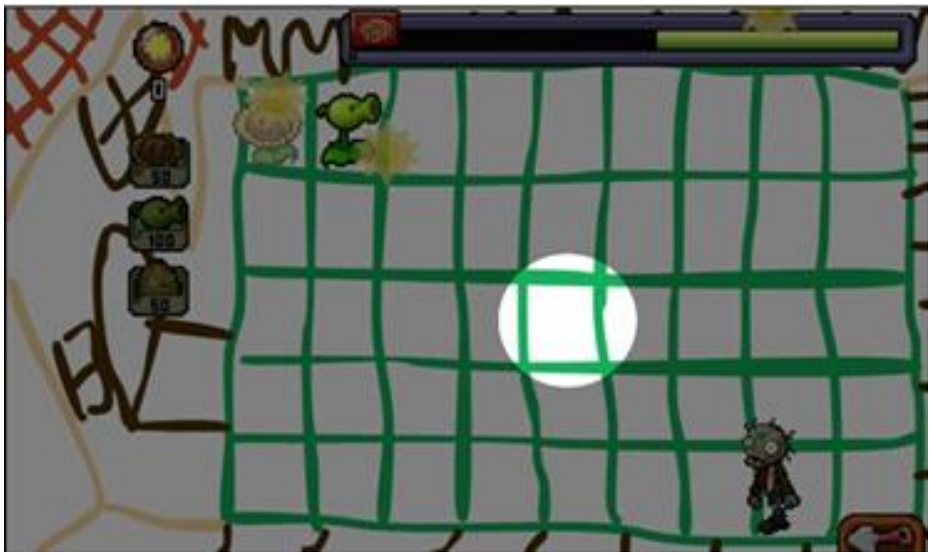
The pea deals damage to enemy from far distance.



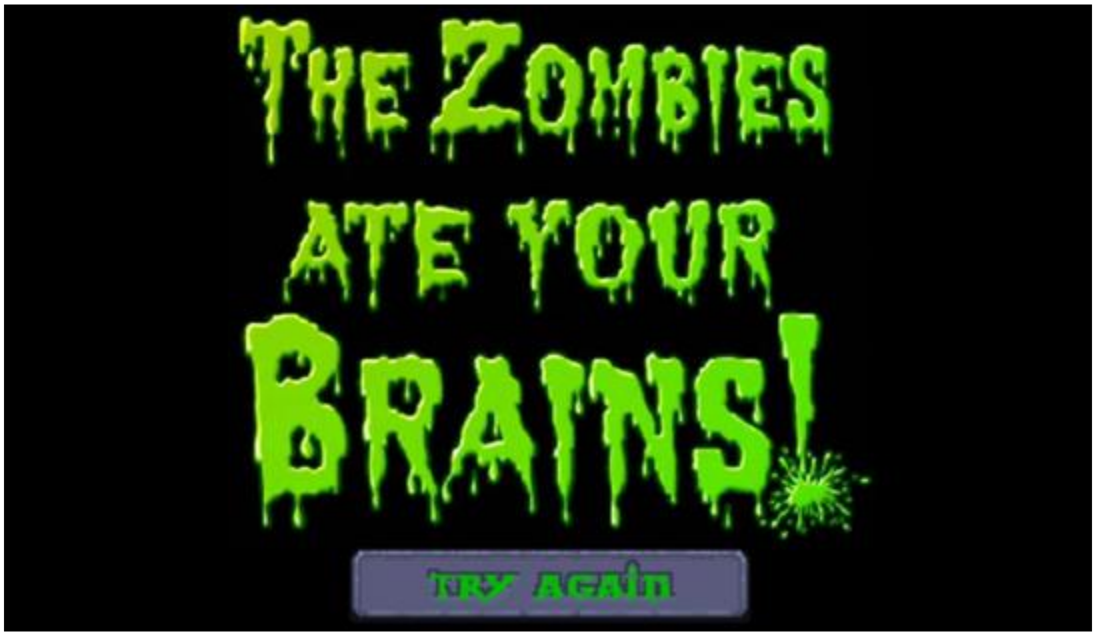
Use the shovel to remove a plant.


50
100
50


50
100
50


50
100
50

When you lose, a screen will appear to say that:



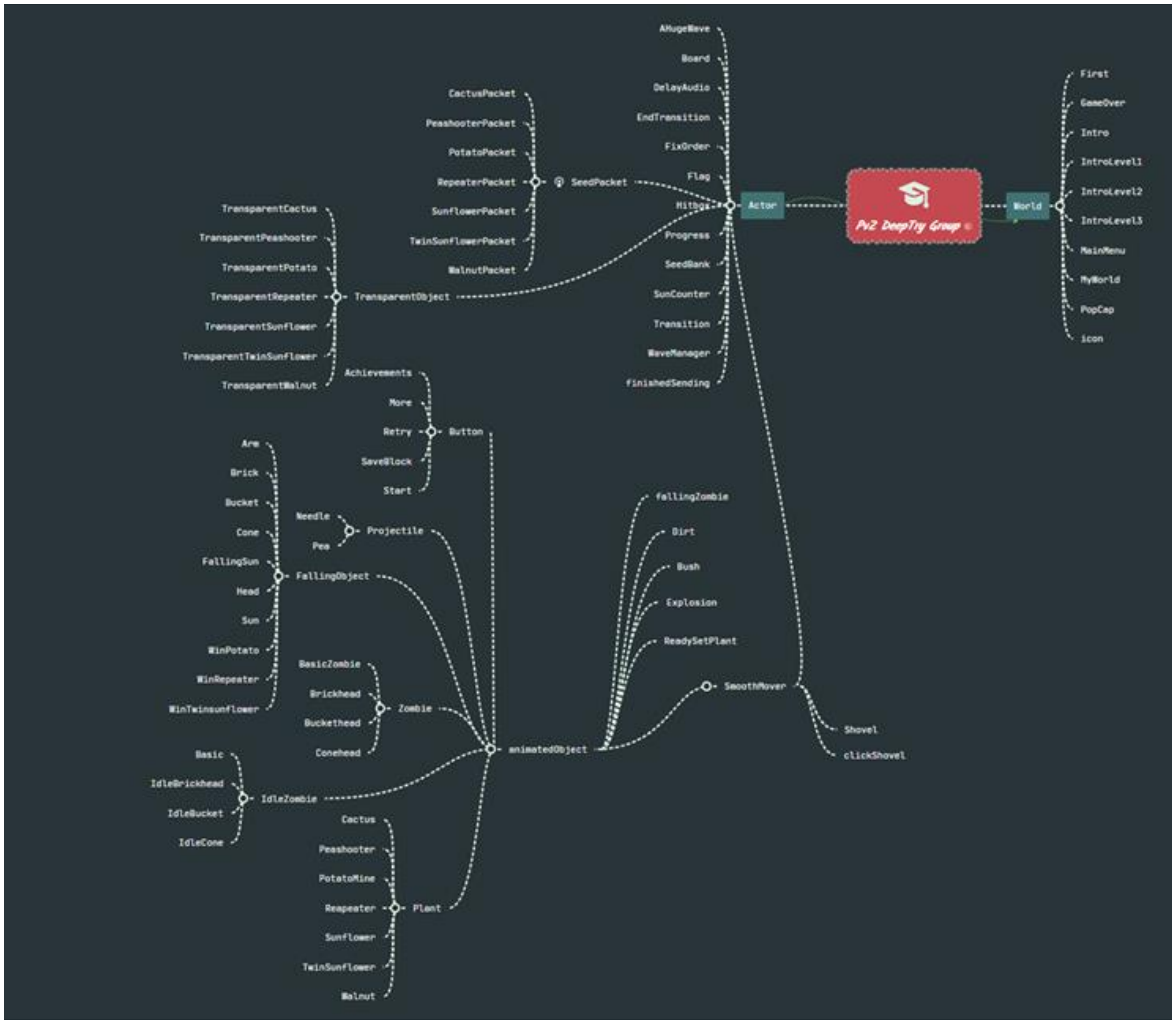
And you can choose to try again.

IV. DESIGN & IMPLEMENTATION

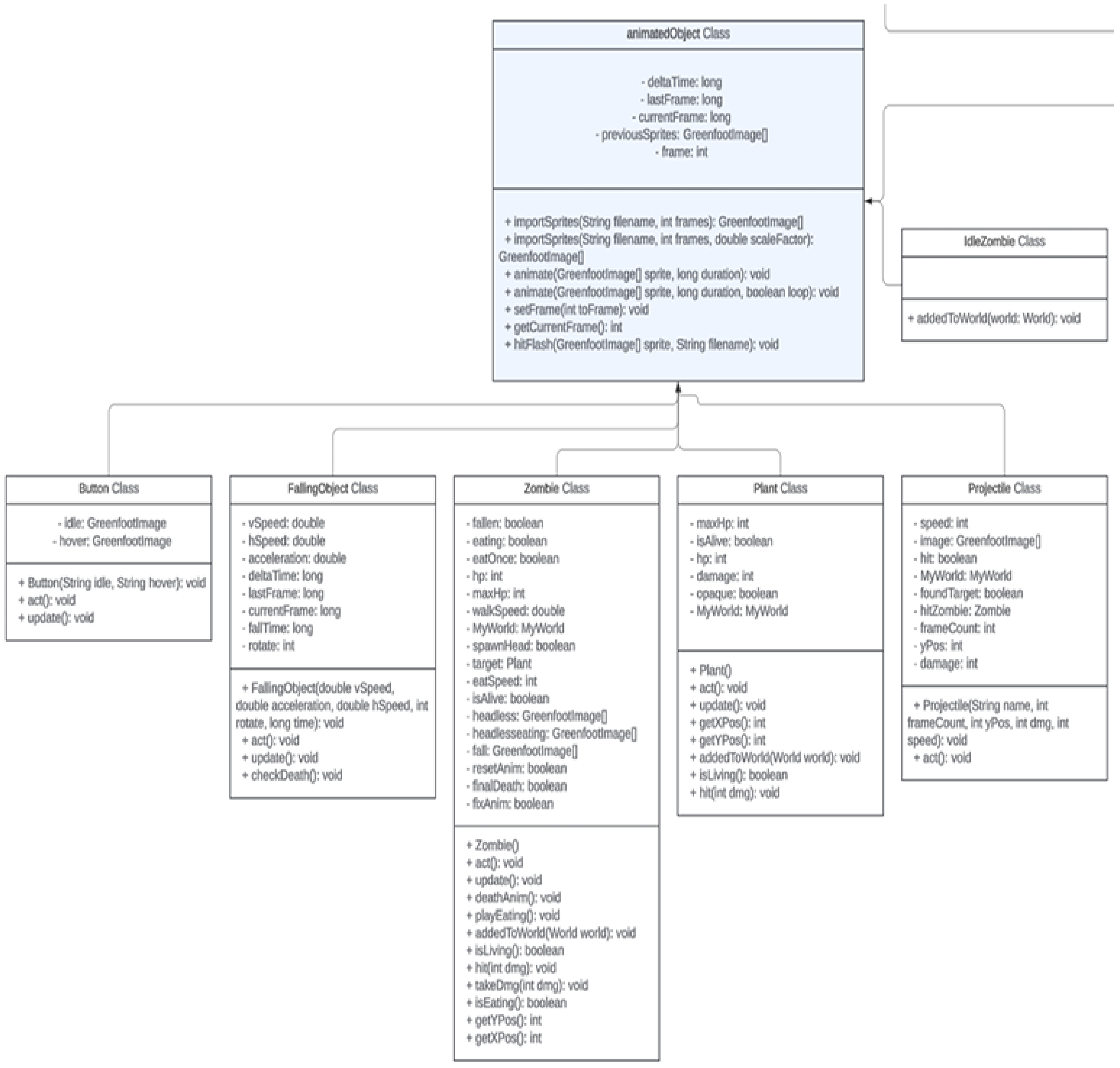
1. UMLs

a. UML Class Diagram

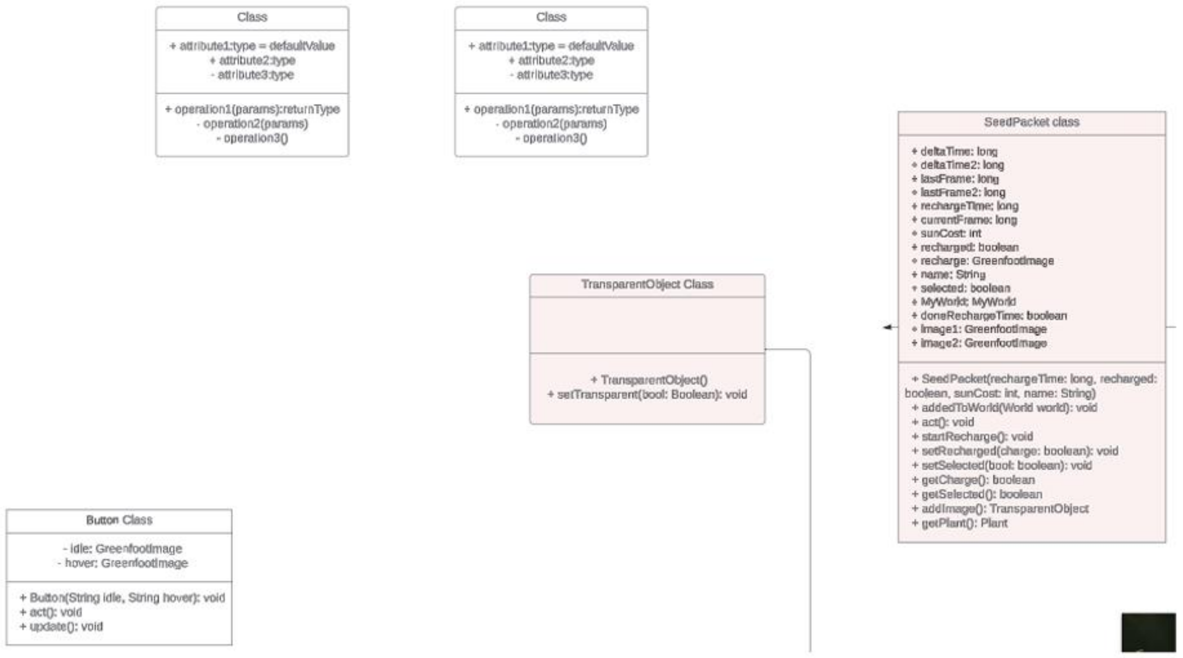
There are 2 main classes: World, Actor.



b. UML of animateObject class



c. UML of SeedPacket class



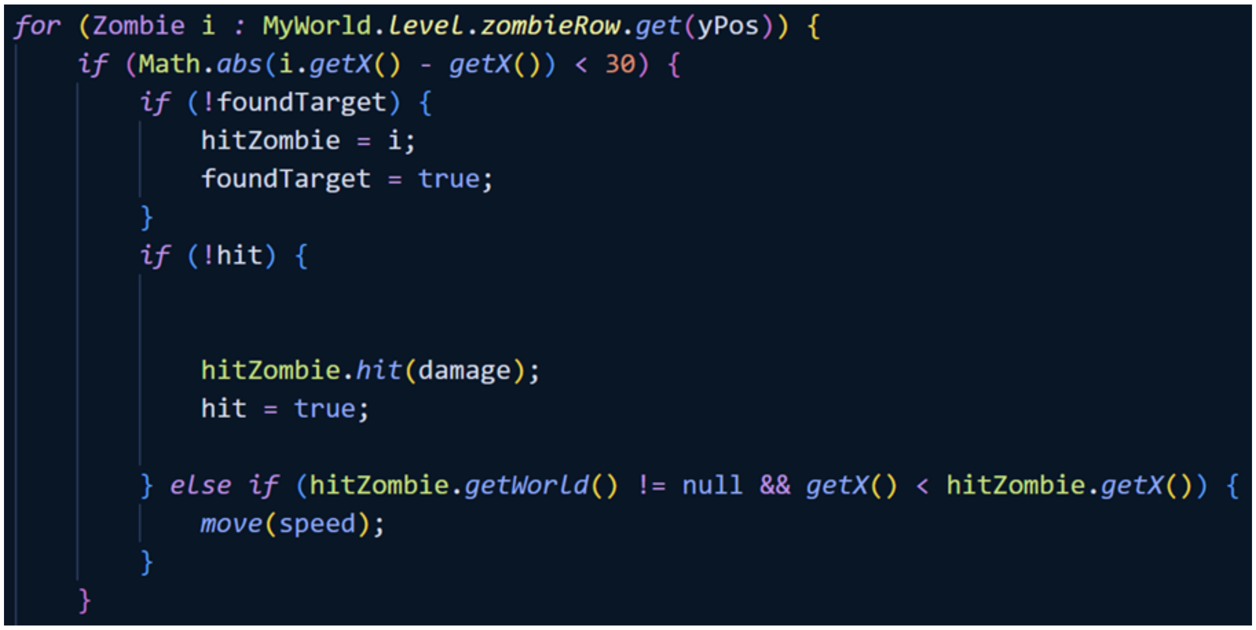
2. Plant class

The "Plants vs. Zombies" game centers around players using various types of plants to fend off waves of zombies. We show off the design, implementation, and functionality of the plants, as well as the class attributes, methods, and inheritance structure used to create diverse plant behaviors.

a. Shooting mechanism

The shooting mechanism in the game is a vital feature, especially for offensive plants like the Peashooter. This report outlines the design, implementation, and functionality of the shooting mechanism, detailing how plants detect zombies, shoot projectiles, and cause damage.

In our game, we use Only “pea” as the only projectile. And the projectiles are built by:



```
for (Zombie i : MyWorld.level.zombieRow.get(yPos)) {
    if (Math.abs(i.getX() - getX()) < 30) {
        if (!foundTarget) {
            hitZombie = i;
            foundTarget = true;
        }
        if (!hit) {


            hitZombie.hit(damage);
            hit = true;

        } else if (hitZombie.getWorld() != null && getX() < hitZombie.getX()) {
            move(speed);
        }
    }
```

b. Specific Plant types

In PvZ, we includes 4 variaties of plants: Shooting (Peashooters), Producing (Sunflowers), Defending (Potato), and Exploding (PotatoMine).

Peashooters: hp = 60, dmg = 10.

Sunflowers: hp = 60, time to produce a sun = 20s (much longer than usual)

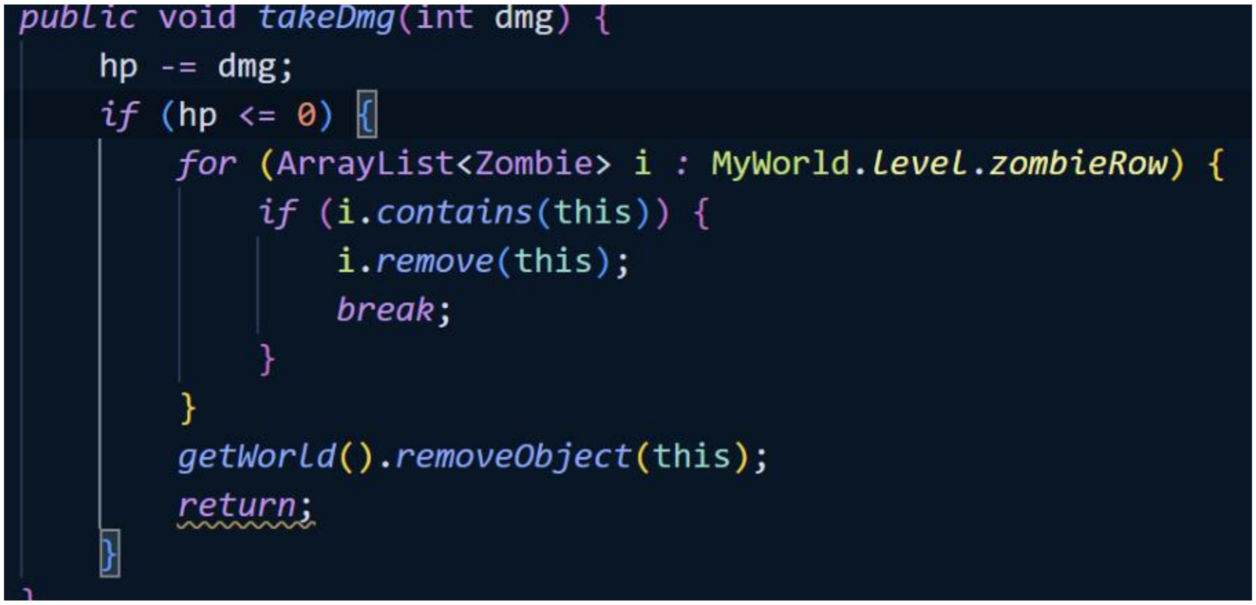
Wallnut: hp = 730 (increased).

PotatoMine: hp (ungrown) = 60, dmg = die.

3. Zombie class

The Zombie class was designed using Object-Oriented Programming (OOP) principles to ensure modularity and reusability. The main class, Zombie, serves as a base class from which specific zombie types inherit.

The hp of zombie is count by:

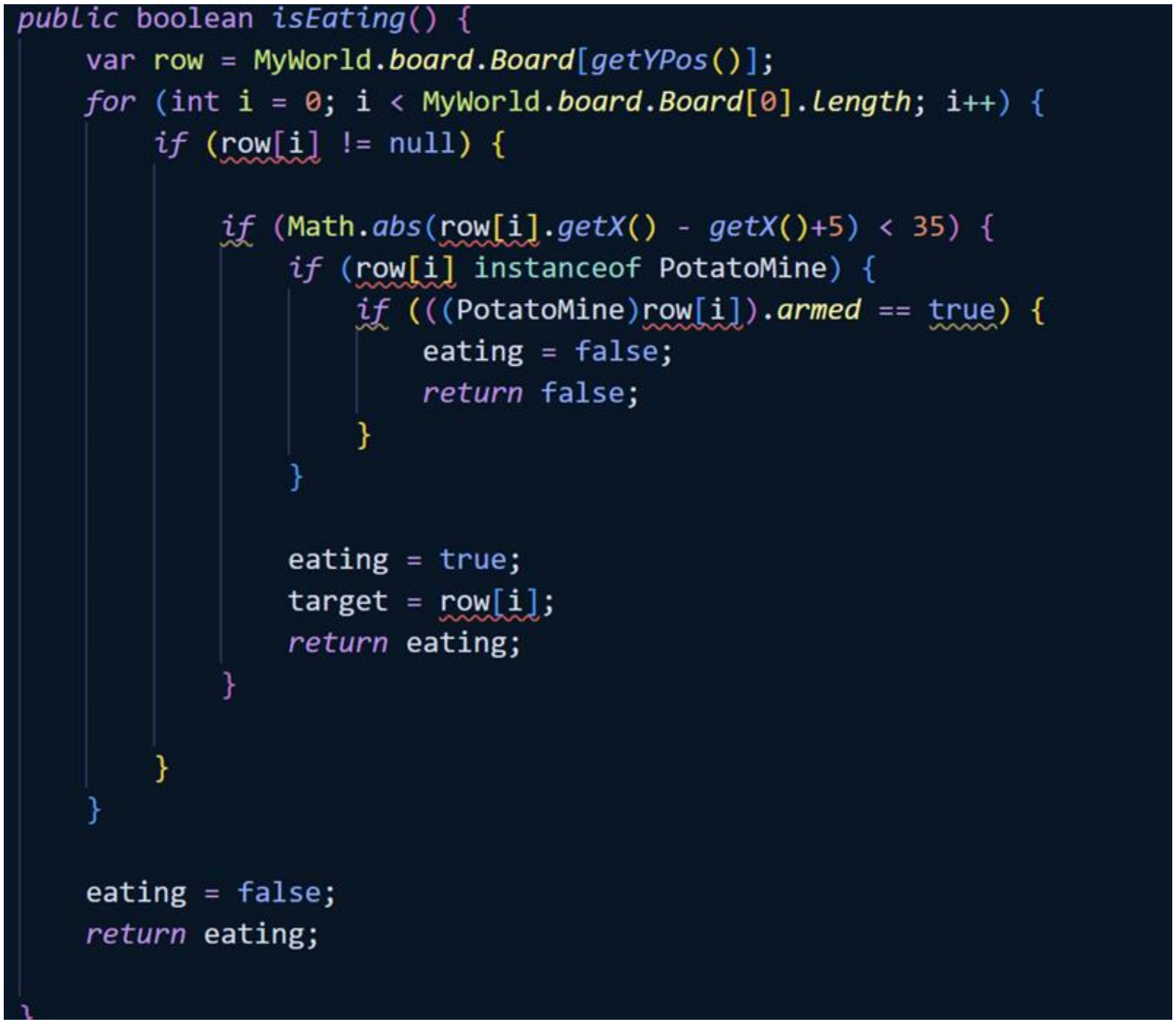


```
public void takeDmg(int dmg) {
    hp -= dmg;
    if (hp <= 0) {
        for (ArrayList<Zombie> i : MyWorld.level.zombieRow) {
            if (i.contains(this)) {
                i.remove(this);
                break;
            }
        }
        getWorld().removeObject(this);
        return;
    }
```

We changed the hp values of zombies to be tankier.

a. Eating mechanism

The eating mechanism is a critical feature that determines how zombies interact with and damage plants. This include the design, implementation, and functionality of the eating mechanism, detailing how zombies move towards and destroy plants.



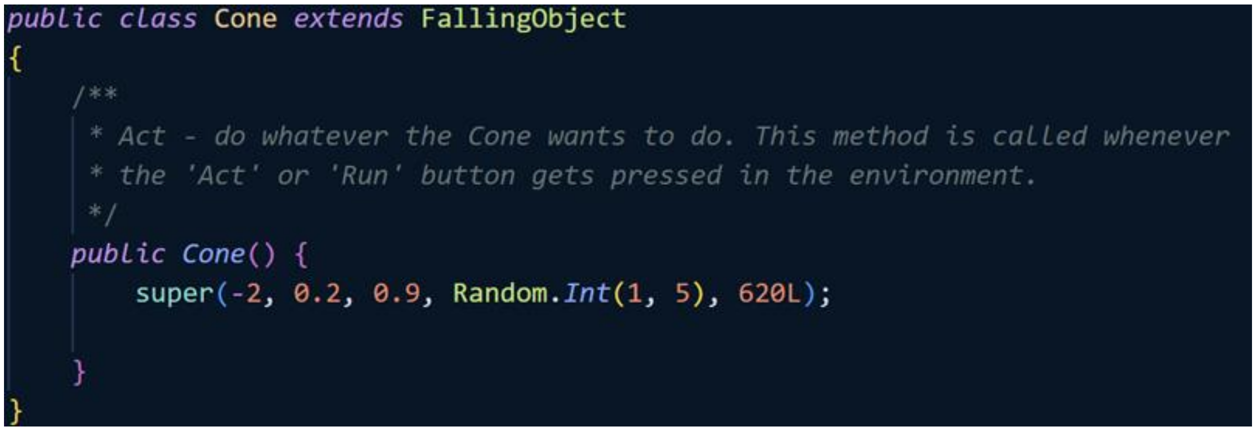
```
public boolean isEating() {
    var row = MyWorld.board.Board[getYPos()];
    for (int i = 0; i < MyWorld.board.Board[0].Length; i++) {
        if (row[i] != null) {

            if (Math.abs(row[i].getX() - getX()+5) < 35) {
                if (row[i] instanceof PotatoMine) {
                    if (((PotatoMine)row[i]).armed == true) {
                        eating = false;
                        return false;
                    }
                }

                eating = true;
                target = row[i];
                return eating;
            }

        }
    }

    eating = false;
    return eating;

}
```

We set the grown potato mine to be inedible.

b. Specific Zombie Types

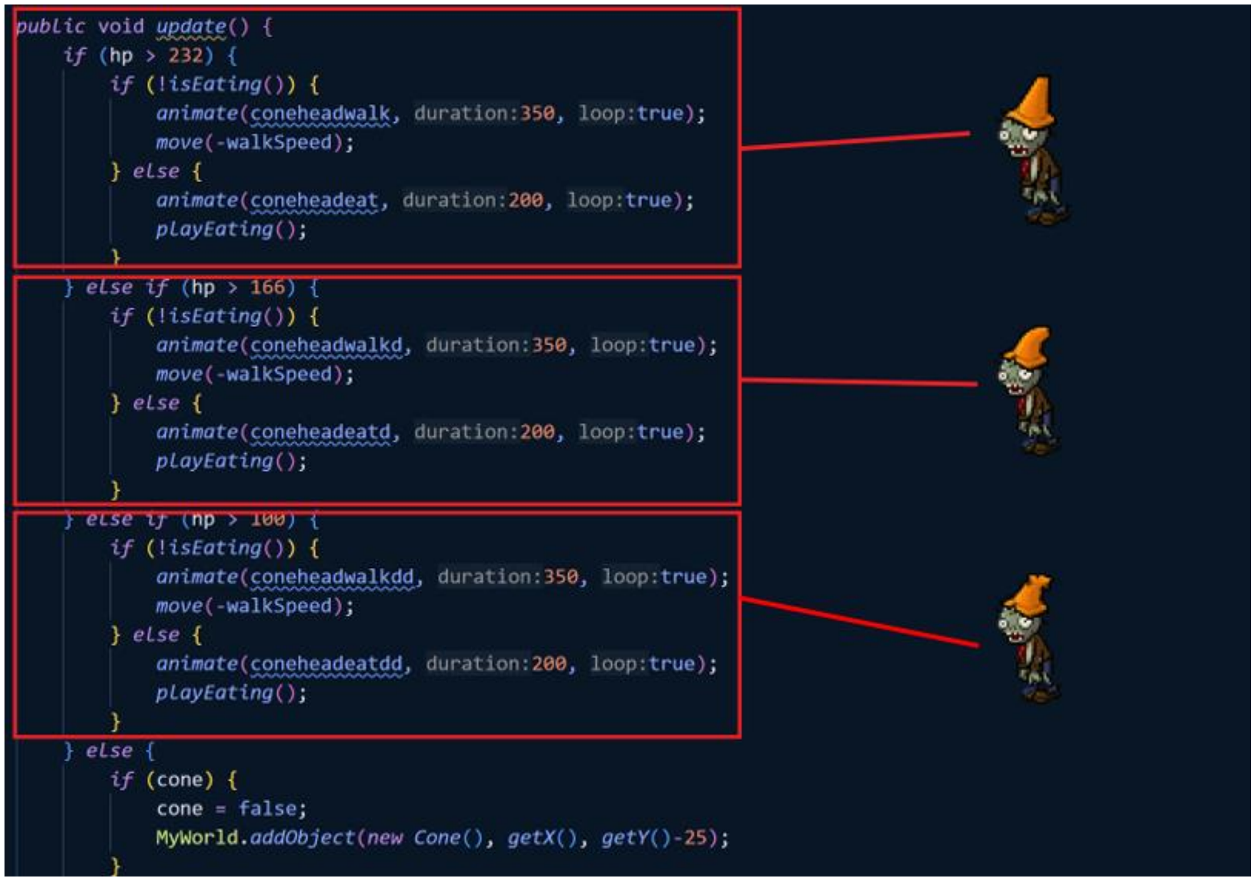
To create diverse zombie behaviors, specific zombie types were derived from the base Zombie class. Each type has unique attributes and methods.

For the zombies with an object on their head, we set the health for them and their specific animations, such as



```java
public class Cone extends FallingObject
{
    /**
     * Act - do whatever the Cone wants to do. This method is called whenever
     * the 'Act' or 'Run' button gets pressed in the environment.
     */
    public Cone() {
        super(-2, 0.2, 0.9, Random.Int(1, 5), 620L);

    }
}
```

This technique also be used with arms and heads. For the cone:

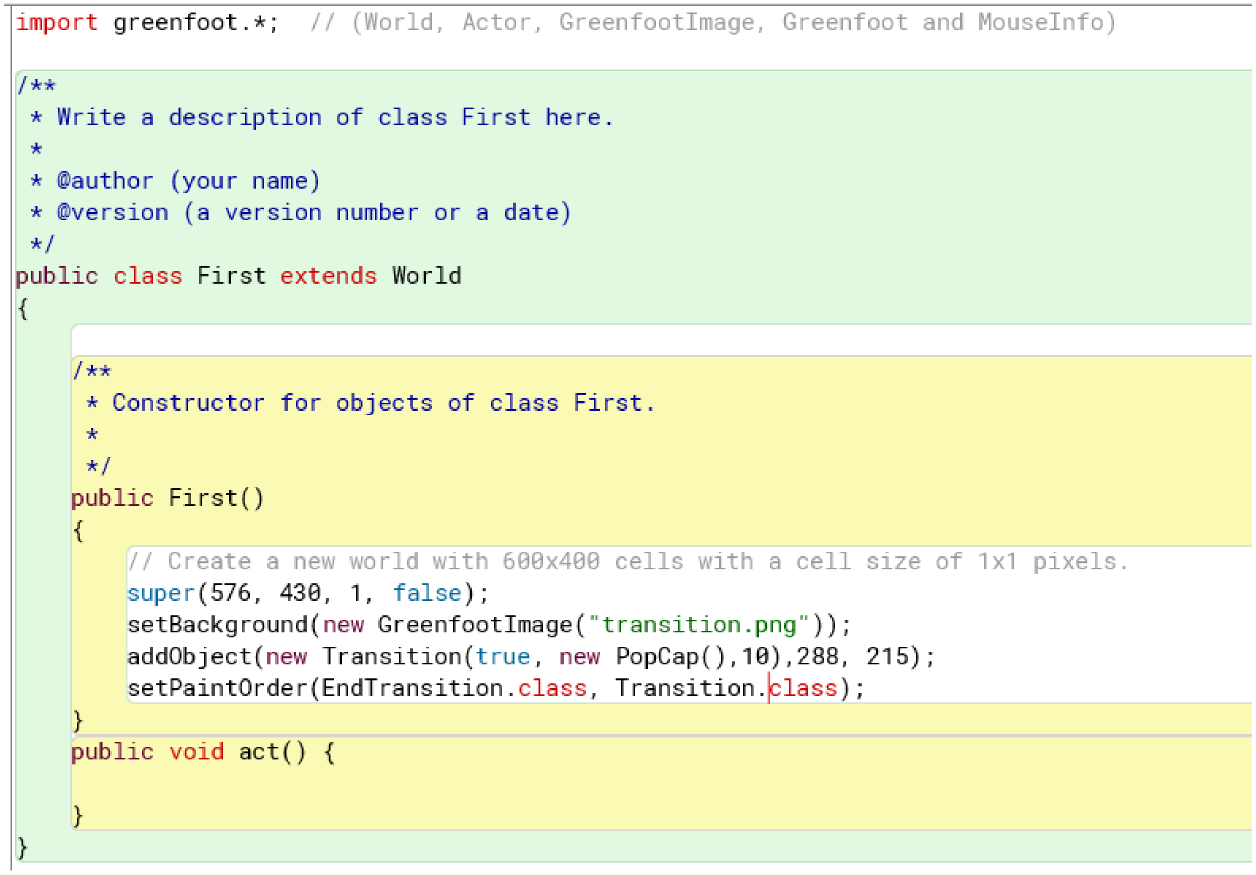


```java
public void update() {
    if (hp > 232) {
        if (!isEating()) {
            animate(coneheadwalk, duration:350, loop:true);
            move(-walkSpeed);
        } else {
            animate(coneheadeat, duration:200, loop:true);
            playEating();
        }
    } else if (hp > 166) {
        if (!isEating()) {
            animate(coneheadwalkd, duration:350, loop:true);
            move(-walkSpeed);
        } else {
            animate(coneheadeatd, duration:200, loop:true);
            playEating();
        }
    } else if (hp > 100) {
        if (!isEating()) {
            animate(coneheadwalkdd, duration:350, loop:true);
            move(-walkSpeed);
        } else {
            animate(coneheadeatdd, duration:200, loop:true);
            playEating();
        }
    } else {
        if (cone) {
            cone = false;
            MyWorld.addObject(new Cone(), getX(), getY()-25);
        }
```

The armored zombies interact with other game components in the same way as standard zombies but with increased health. Their higher durability makes them more challenging to defeat, but with the same technique.

V. Description

When first entering the game we see the black screen call: First class extends from World.



```java
import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

/**
 * Write a description of class First here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class First extends World
{

    /**
     * Constructor for objects of class First.
     *
     */
    public First()
    {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(576, 430, 1, false);
        setBackground(new GreenfootImage("transition.png"));
        addObject(new Transition(true, new PopCap(),10),288, 215);
        setPaintOrder(EndTransition.class, Transition.class);
    }
    public void act() {

    }
}
```

- **super(576, 430, 1, false)**:
    - Creates a new world with *dimensions 576x430 pixels.*
    - The grid cells have *a size of 1x1 pixel.*
    - The false parameter specifies that the world does not use *a background grid (the grid is invisible).*
- **setBackground(new GreenfootImage("transition.png"))**:
    - *Sets the background image for the world to transition.png*.
- **addObject(new Transition(true, new PopCap(), 10), 288, 215)**:
    - *Adds an instance of the Transition class* to the world at coordinates (288, 215).

    - This likely represents a transition effect or a moving object that facilitates the change to another world (in this case, PopCap).
    - The Transition object seems to take parameters, with true perhaps indicating a forward direction, new PopCap() specifying the world to transition to, and 10 possibly setting some transition speed or effect.

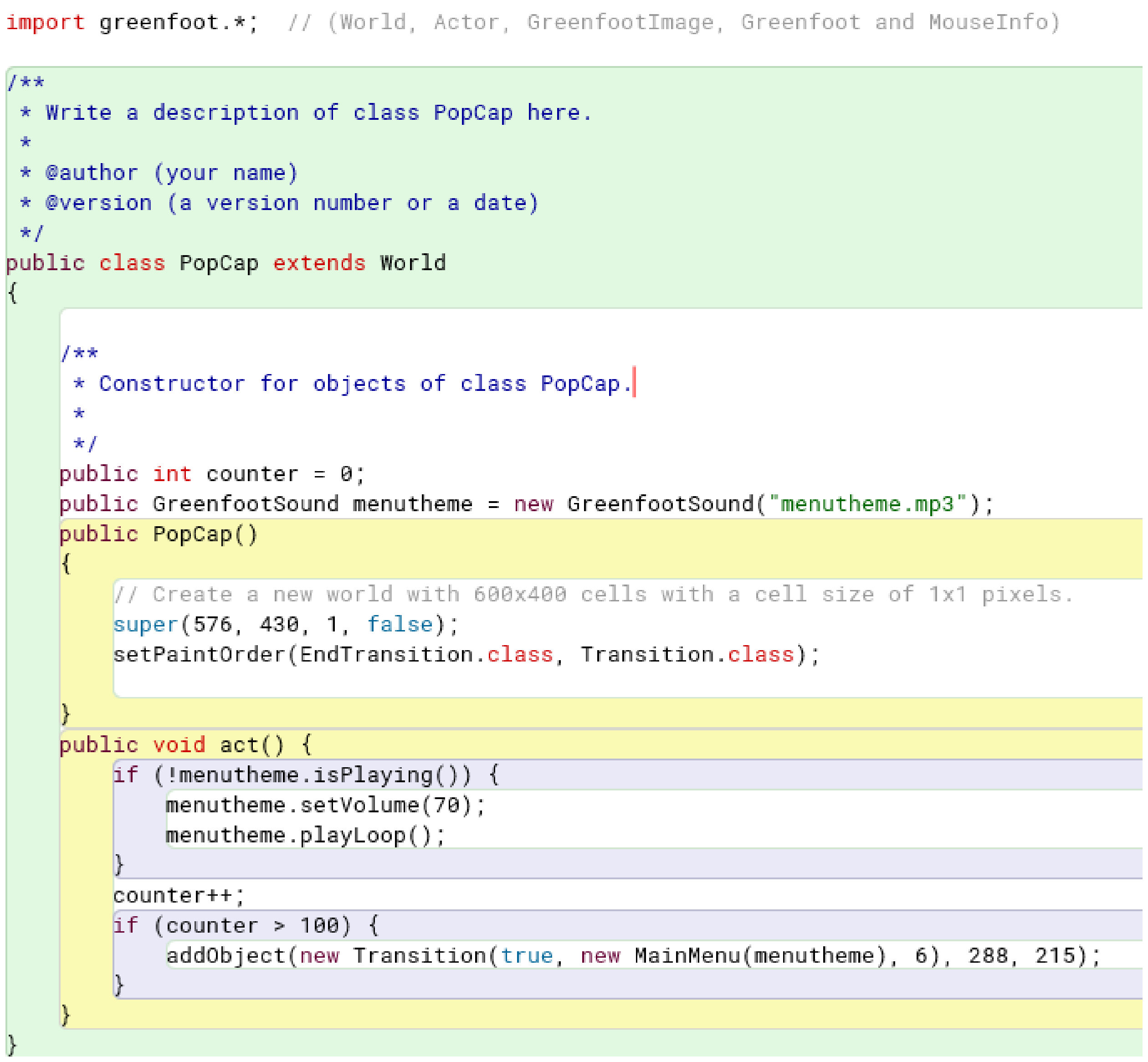
- **setPaintOrder(EndTransition.class, Transition.class)**:
    - Specifies *the order in which objects are drawn*. Here, it ensures that EndTransition objects are drawn above Transition objects, which could control the visual layering of elements during the transition.

So we can see that it will turn the music on together with showing the PopCap logo (Music Playback, Frame Counter, Transition).

GreenfootSound class representing background music and the file menutheme.mp3 is loaded to be played in the opening world until the game starts, the music theme will change.

This Greenfoot Java class defines a PopCap world, which serves as a game screen with specific behaviors and properties.

PopCap extends the World class provided by Greenfoot, meaning it represents a screen or "world" in the game.



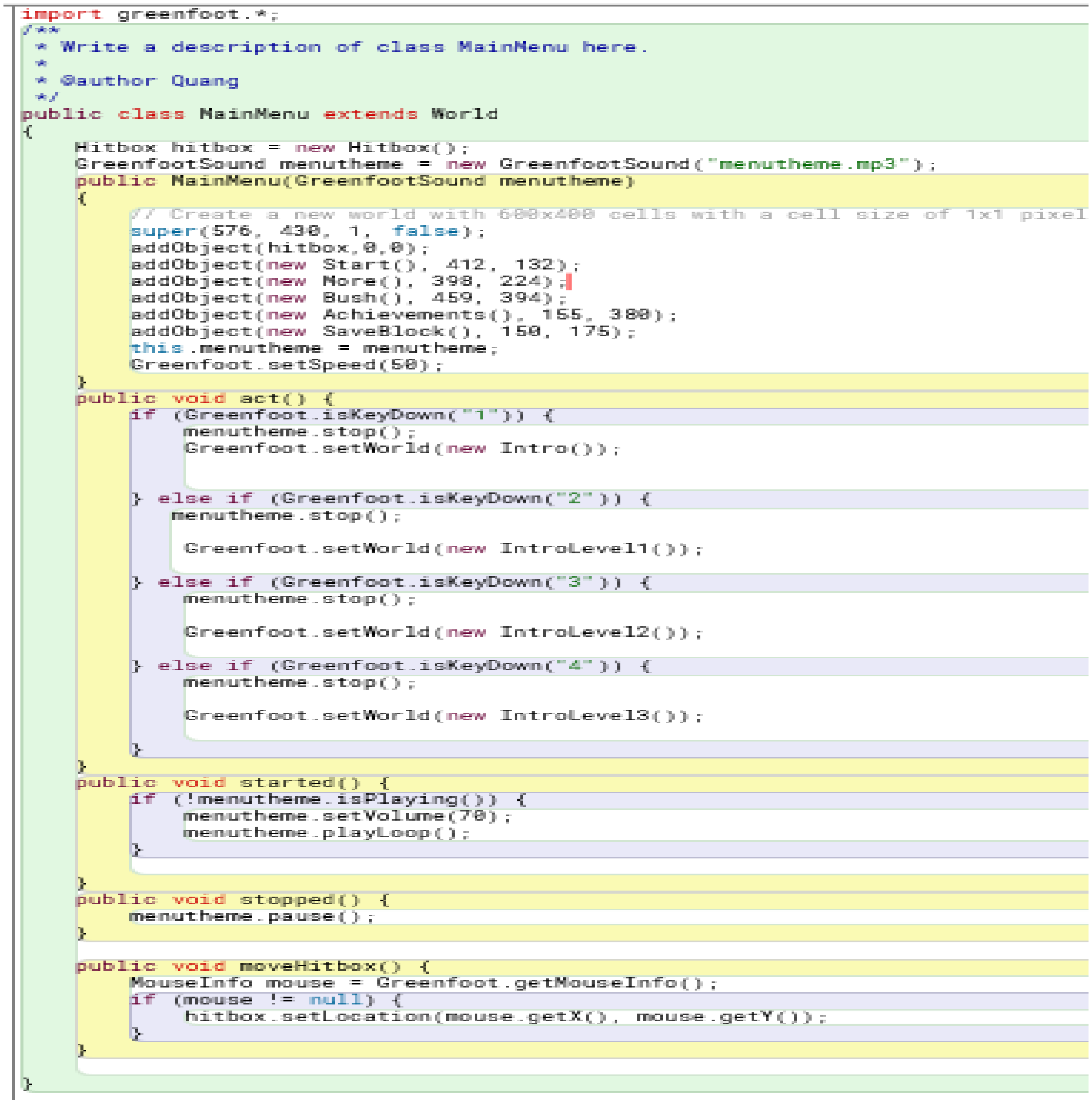
```java
import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

/**
 * Write a description of class PopCap here.
 * 
 * @author (your name) 
 * @version (a version number or a date)
 */
public class PopCap extends World
{

    /**
     * Constructor for objects of class PopCap.
     * 
     */
    public int counter = 0;
    public GreenfootSound menutheme = new GreenfootSound("menutheme.mp3");
    public PopCap()
    {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(576, 430, 1, false);
        setPaintOrder(EndTransition.class, Transition.class);

    }
    public void act() {
        if (!menutheme.isPlaying()) {
            menutheme.setVolume(70);
            menutheme.playLoop();
        }
        counter++;
        if (counter > 100) {
            addObject(new Transition(true, new MainMenu(menutheme), 6), 288, 215);
        }
    }
}
```

A counter used to track how many frames have passed since the world was created.

The code Initializes the world with a size of **576x430 pixels**, where each cell is 1x1 pixel. Disables "bounded scrolling" by passing false to the super constructor. Sets the order of rendering (setPaintOrder) so that instances of EndTransition are drawn on top of Transition.

Transition class to manages screen transitions. Mainmenu class is to represent the next world or menu screen. Endtransition class is a visual element rendered above transition objects for the transition effect.

Then we can see the mainmenu.



```java
import greenfoot.*;
/**
 * Write a description of class MainMenu here.
 *
 * @author Quang
 */
public class MainMenu extends World
{
    Hitbox hitbox = new Hitbox();
    GreenfootSound menutheme = new GreenfootSound("menutheme.mp3");
    public MainMenu(GreenfootSound menutheme)
    {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixel
        super(576, 430, 1, false);
        addObject(hitbox,0,0);
        addObject(new Start(), 412, 132);
        addObject(new More(), 398, 224);
        addObject(new Bush(), 459, 394);
        addObject(new Achievements(), 155, 380);
        addObject(new SaveBlock(), 150, 175);
        this.menutheme = menutheme;
        Greenfoot.setSpeed(50);
    }
    public void act() {
        if (Greenfoot.isKeyDown("1")) {
            menutheme.stop();
            Greenfoot.setWorld(new Intro());

        } else if (Greenfoot.isKeyDown("2")) {
           menutheme.stop();

            Greenfoot.setWorld(new IntroLevel1());
        } else if (Greenfoot.isKeyDown("3")) {
            menutheme.stop();

            Greenfoot.setWorld(new IntroLevel2());
        } else if (Greenfoot.isKeyDown("4")) {
            menutheme.stop();

            Greenfoot.setWorld(new IntroLevel3());
        }
    }
    public void started() {
        if (!menutheme.isPlaying()) {
            menutheme.setVolume(70);
            menutheme.playLoop();
        }

    }
    public void stopped() {
        menutheme.pause();
    }

    public void moveHitbox() {
        MouseInfo mouse = Greenfoot.getMouseInfo();
        if (mouse != null) {
            hitbox.setLocation(mouse.getX(), mouse.getY());
        }
    }

}
```

- **World Setup**:
    - The world is created with a size of *576x430 pixels*.
    - *The menutheme sound* is passed in, which is the background music for the menu.
- **Objects Added to the World**:
    - **hitbox**: A Hitbox object is added to *track user input* (likely for detecting mouse clicks or hover events).

- **Start, More, Bush, Achievements, SaveBlock**: These are interactive objects added to the world at specific coordinates. These objects likely represent buttons or clickable areas within the menu for user interaction.

- **Sound and Speed**:
    - The menutheme background music is assigned and will be played when the menu is active.
    - The game speed is set to 50 (likely controlling the frame rate).

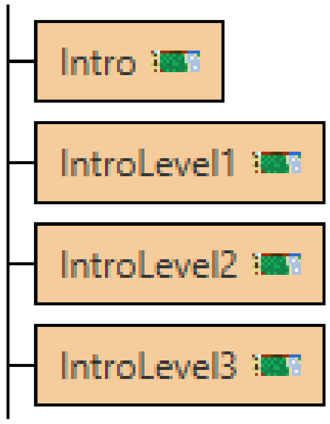
Method: act()

- *This method checks for key presses (1, 2, 3, 4)* and transitions the world based on which key is pressed.
- Each key corresponds to a different level or section of the game (e.g., Intro, IntroLevel1, etc.). It also stops the background music before transitioning to a new world.

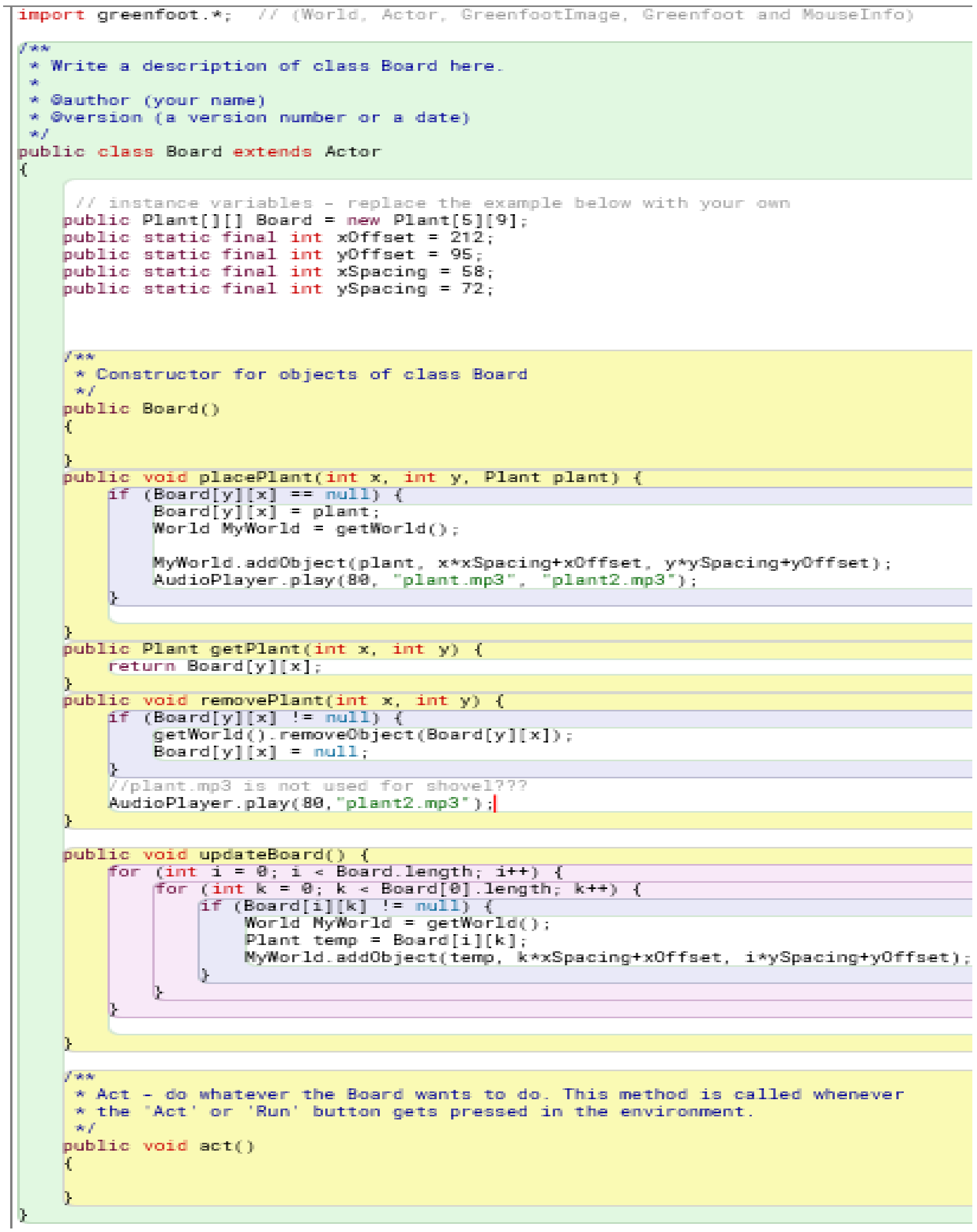
Methods for Handling Music and Hitbox Movement:

- **started()**:
    - Ensures the background music starts playing in a loop when the world begins.
- **stopped()**:
    - Pauses the background music when the world stops.
- **moveHitbox()**:
    - Moves the hitbox to follow the mouse cursor. This allows for visual feedback of the area the player is interacting wit

There are 4 level with respect to the key presses (1, 2, 3, 4)



This Java code defines a Board class for a Greenfoot project.



```
import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

/**
 * Write a description of class Board here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Board extends Actor
{
    // instance variables - replace the example below with your own
    public Plant[][] Board = new Plant[5][9];
    public static final int xOffset = 212;
    public static final int yOffset = 95;
    public static final int xSpacing = 58;
    public static final int ySpacing = 72;


    /**
     * Constructor for objects of class Board
     */
    public Board()
    {

    }
    public void placePlant(int x, int y, Plant plant) {
        if (Board[y][x] == null) {
            Board[y][x] = plant;
            World MyWorld = getWorld();

            MyWorld.addObject(plant, x*xSpacing+xOffset, y*ySpacing+yOffset);
            AudioPlayer.play(80, "plant.mp3", "plant2.mp3");
        }

    }
    public Plant getPlant(int x, int y) {
        return Board[y][x];
    }
    public void removePlant(int x, int y) {
        if (Board[y][x] != null) {
            getWorld().removeObject(Board[y][x]);
            Board[y][x] = null;
        }
        //plant.mp3 is not used for shovel???
        AudioPlayer.play(80,"plant2.mp3");
    }

    public void updateBoard() {
        for (int i = 0; i < Board.length; i++) {
            for (int k = 0; k < Board[0].length; k++) {
                if (Board[i][k] != null) {
                    World MyWorld = getWorld();
                    Plant temp = Board[i][k];
                    MyWorld.addObject(temp, k*xSpacing+xOffset, i*ySpacing+yOffset);
                }
            }
        }

    }

    /**
     * Act - do whatever the Board wants to do. This method is called whenever
     * the 'Act' or 'Run' button gets pressed in the environment.
     */
    public void act()
    {

    }
}
```

The class is part of a grid-based game (possibly inspired by games like Plants vs. Zombies).

The Board class extends Actor, meaning it represents an object that can interact within a World.

placePlant(int x, int y, Plant plant)

- **Purpose**: Places a Plant object at the specified grid coordinates (x, y) if the cell is empty

Steps:

1. Checks if Board[y][x] is null (i.e., the cell is empty).
2. If empty:
    - Assigns the Plant object to the grid cell.
    - Adds the Plant object to the world at a calculated position:
        - x * xSpacing + xOffset: X-coordinate in pixels.
        - y * ySpacing + yOffset: Y-coordinate in pixels.
3. Plays an audio clip (plant.mp3 or plant2.mp3) using AudioPlayer.play.

getPlant(int x, int y)

- **Purpose**: Returns the Plant object at the specified grid cell (x, y).
- **Steps**:
    - Simply retrieves and returns the value of Board[y][x].

removePlant(int x, int y)

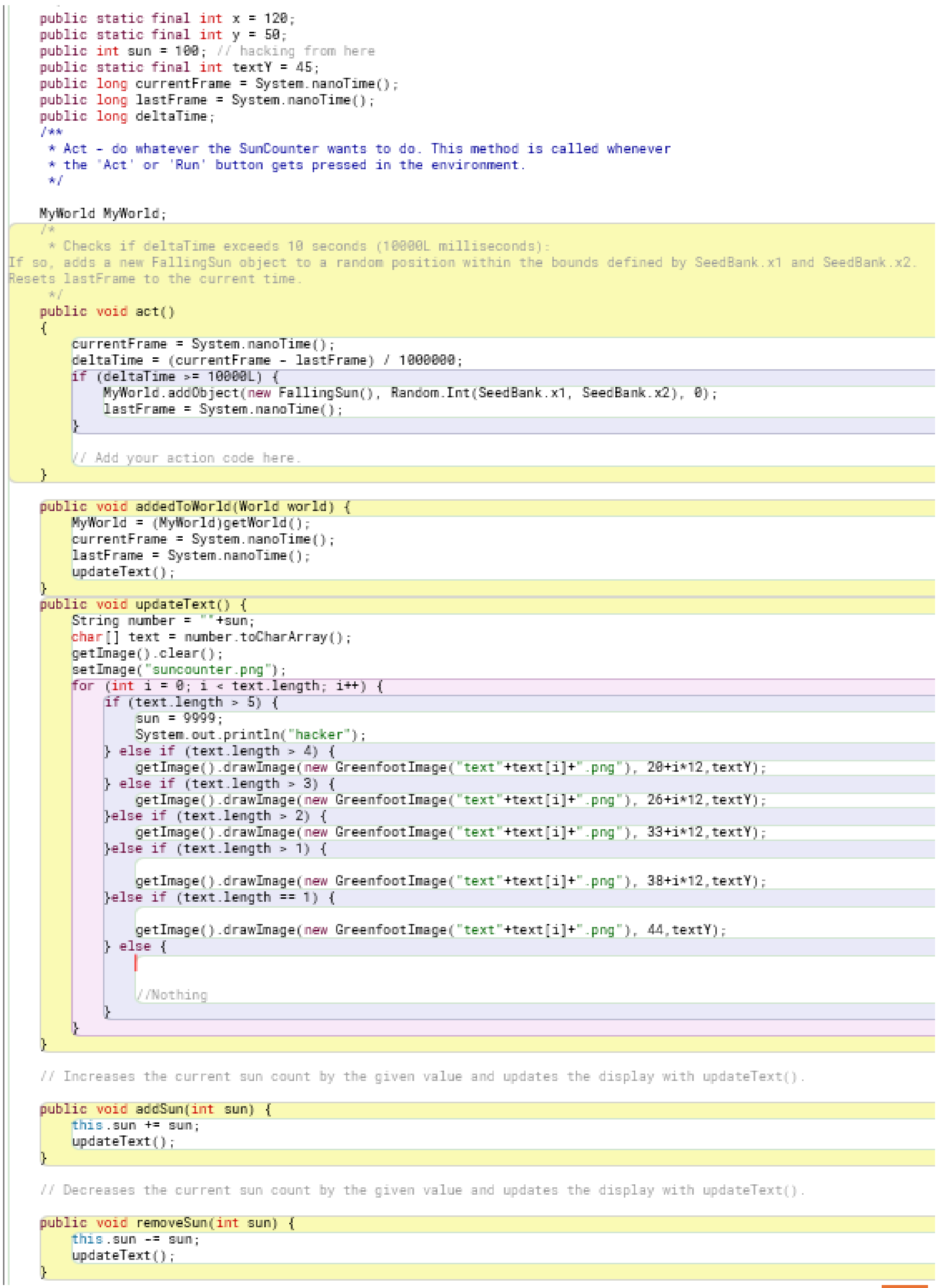
- **Purpose**: Removes the Plant object from the specified grid cell (x, y).
- **Steps**:
    1. Checks if the cell contains a Plant.
    2. If so:
        - Removes the Plant object from the World.
        - Sets Board[y][x] to null (indicating the cell is now empty).

3. Plays an audio clip (plant2.mp3) via AudioPlayer.

updateBoard()

- **Purpose**: Ensures the visual placement of Plant objects matches the grid array.
- **Steps**:
    1. Iterates through the Board array.
    2. For every non-null cell, retrieves the Plant object.
    3. Adds the Plant to the World at the calculated screen position:
        - X: k * xSpacing + xOffset
        - Y: i * ySpacing + yOffset

The SunCounter class in this code represents the player's "sun" currency in the game. It provides functionality for displaying and managing the player's sun count and also periodically spawns new falling sun objects on the screen.



```
    public static final int x = 120;
    public static final int y = 50;
    public int sun = 100; // hacking from here
    public static final int textY = 45;
    public long currentFrame = System.nanoTime();
    public long lastFrame = System.nanoTime();
    public long deltaTime;
    /**
     * Act - do whatever the SunCounter wants to do. This method is called whenever
     * the 'Act' or 'Run' button gets pressed in the environment.
     */

    MyWorld MyWorld;
    /*
     * Checks if deltaTime exceeds 10 seconds (10000L milliseconds):
If so, adds a new FallingSun object to a random position within the bounds defined by SeedBank.x1 and SeedBank.x2.
Resets lastFrame to the current time.
     */
    public void act()
    {
        currentFrame = System.nanoTime();
        deltaTime = (currentFrame - lastFrame) / 1000000;
        if (deltaTime >= 10000L) {
            MyWorld.addObject(new FallingSun(), Random.Int(SeedBank.x1, SeedBank.x2), 0);
            lastFrame = System.nanoTime();
        }

        // Add your action code here.
    }

    public void addedToWorld(World world) {
        MyWorld = (MyWorld)getWorld();
        currentFrame = System.nanoTime();
        lastFrame = System.nanoTime();
        updateText();
    }
    public void updateText() {
        String number = ""+sun;
        char[] text = number.toCharArray();
        getImage().clear();
        setImage("suncounter.png");
        for (int i = 0; i < text.length; i++) {
            if (text.length > 5) {
                sun = 9999;
                System.out.println("hacker");
            } else if (text.length > 4) {
                getImage().drawImage(new GreenfootImage("text"+text[i]+".png"), 20+i*12,textY);
            } else if (text.length > 3) {
                getImage().drawImage(new GreenfootImage("text"+text[i]+".png"), 26+i*12,textY);
            }else if (text.length > 2) {
                getImage().drawImage(new GreenfootImage("text"+text[i]+".png"), 33+i*12,textY);
            }else if (text.length > 1) {

                getImage().drawImage(new GreenfootImage("text"+text[i]+".png"), 38+i*12,textY);
            }else if (text.length == 1) {

                getImage().drawImage(new GreenfootImage("text"+text[i]+".png"), 44,textY);
            } else {


                //Nothing
            }
        }
    }

    // Increases the current sun count by the given value and updates the display with updateText().

    public void addSun(int sun) {
        this.sun += sun;
        updateText();
    }

    // Decreases the current sun count by the given value and updates the display with updateText().

    public void removeSun(int sun) {
        this.sun -= sun;
        updateText();
    }
```

Key Attributes:

1. **x, y**:
   - Define the position of the SunCounter object in the world.
2. **sun**:
   - The player's current sun count, initialized to 9998.

```java
public static final int x = 120;
public static final int y = 50;
public int sun = 9998; // hacking from here
public static final int textY = 45;
public long currentFrame = System.nanoTime();
```

3. **textY**:
   - The vertical offset for displaying text.
4. **currentFrame, lastFrame, deltaTime**:
   - Variables for measuring time intervals in the game, used for spawning new falling suns.

Constructor-Like Methods:

- **addedToWorld(World world)**:
  - Called when the SunCounter object is added to the game world.
  - Initializes timing variables and updates the sun count display.

Core Methods:

1. act():

- Runs repeatedly during the game.
- Measures time elapsed since the last update using System.nanoTime() and spawns a FallingSun object every 10 seconds.
- Spawns suns at random horizontal positions between SeedBank.x1 and SeedBank.x2.

---

2. updateText():

- Updates the displayed sun count:
  - Converts the sun count (sun) to a string.
  - Clears the current image and sets it to a base image (suncounter.png).
  - Draws the digits of the sun count using images corresponding to each digit.
  - Adjusts the spacing dynamically based on the number of digits in the sun count.
- **Special Handling**:
  - Limits the sun count to 9999 to prevent overflow or hacking.
  - Prints "hacker" to the console if the count exceeds 5 digits.

---

3. addSun(int sun):

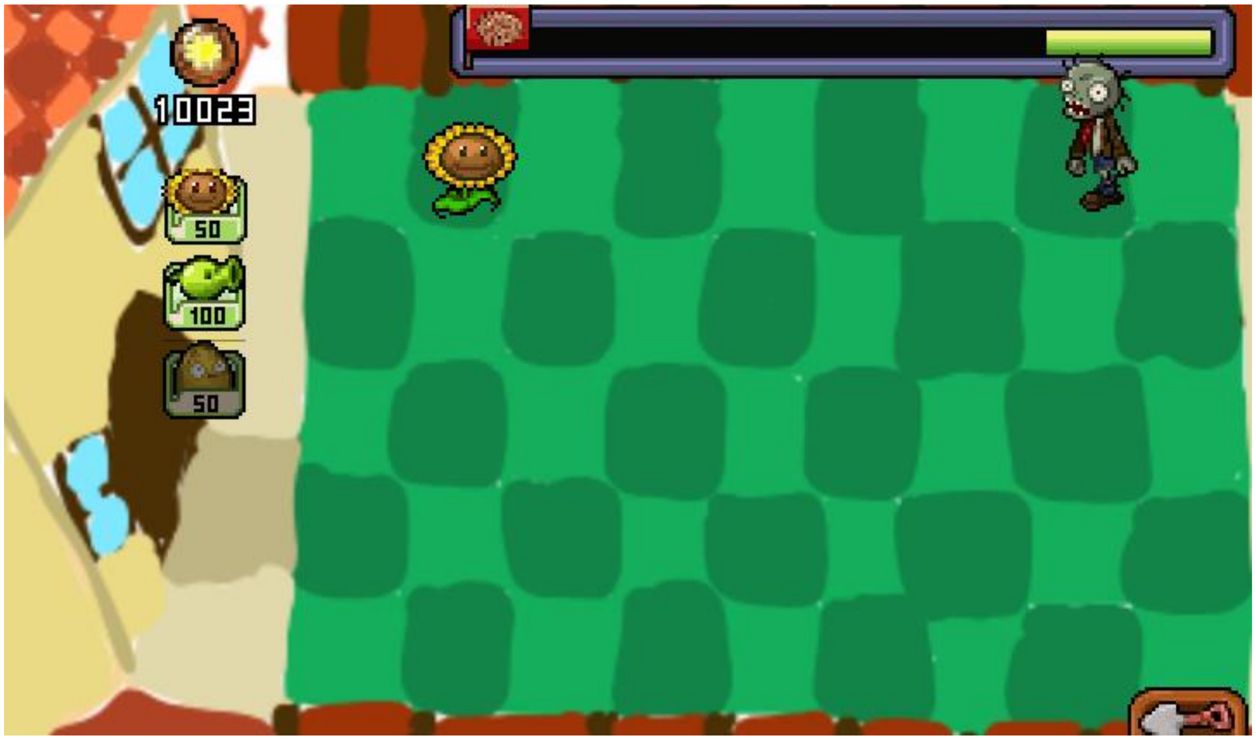
- Increases the player's sun count by the specified value and updates the display.

4. removeSun(int sun):

- Decreases the player's sun count by the specified value and updates the display.

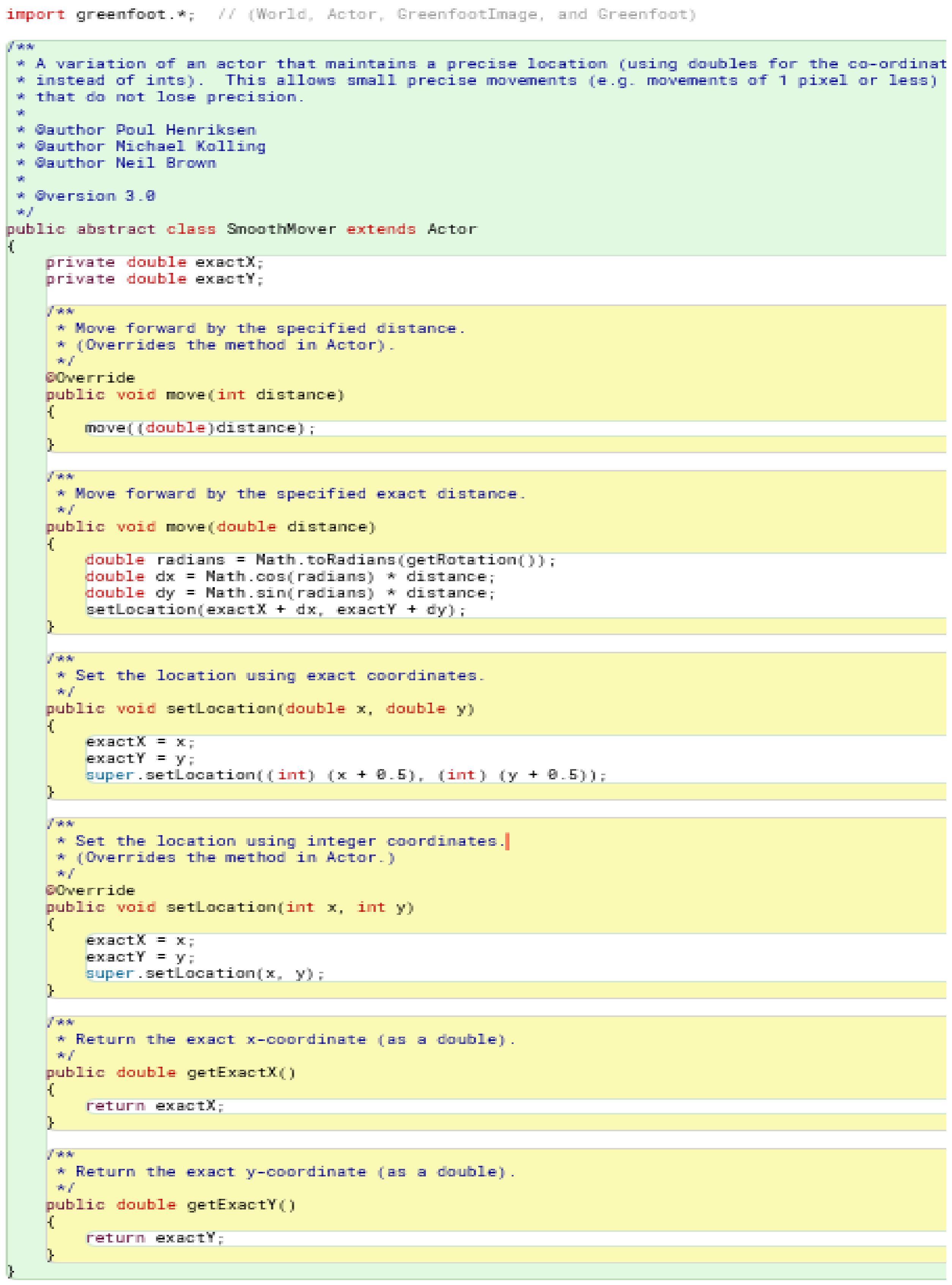
Key Features:

1. **Dynamic Text Rendering**:
   - The sun count is visually represented with dynamically spaced images for each digit.
2. **Time-Based Sun Spawning**:
   - Spawns a FallingSun object every 10 seconds at random positions, enhancing the gameplay dynamic.
3. **Cheat Prevention**:
   - Caps the maximum sun count to 9999 to prevent extreme or invalid values. And it still continue the game but it has the log “hacker”



Cheating with initial value = 9998 sun.

How to make a zombie goes or animated it.



```java
import greenfoot.*;  // (World, Actor, GreenfootImage, and Greenfoot)

/**
 * A variation of an actor that maintains a precise location (using doubles for the co-ordinat
 * instead of ints).  This allows small precise movements (e.g. movements of 1 pixel or less)
 * that do not lose precision.
 * 
 * @author Poul Henriksen
 * @author Michael Kolling
 * @author Neil Brown
 * 
 * @version 3.0
 */
public abstract class SmoothMover extends Actor
{
    private double exactX;
    private double exactY;

    /**
     * Move forward by the specified distance.
     * (Overrides the method in Actor).
     */
    @Override
    public void move(int distance)
    {
        move((double)distance);
    }

    /**
     * Move forward by the specified exact distance.
     */
    public void move(double distance)
    {
        double radians = Math.toRadians(getRotation());
        double dx = Math.cos(radians) * distance;
        double dy = Math.sin(radians) * distance;
        setLocation(exactX + dx, exactY + dy);
    }

    /**
     * Set the location using exact coordinates.
     */
    public void setLocation(double x, double y)
    {
        exactX = x;
        exactY = y;
        super.setLocation((int) (x + 0.5), (int) (y + 0.5));
    }

    /**
     * Set the location using integer coordinates.
     * (Overrides the method in Actor.)
     */
    @Override
    public void setLocation(int x, int y)
    {
        exactX = x;
        exactY = y;
        super.setLocation(x, y);
    }

    /**
     * Return the exact x-coordinate (as a double).
     */
    public double getExactX()
    {
        return exactX;
    }

    /**
     * Return the exact y-coordinate (as a double).
     */
    public double getExactY()
    {
        return exactY;
    }
}
```

Attributes:

1. **exactX**:
   - Tracks the exact horizontal position of the actor as a double.
2. **exactY**:
   - Tracks the exact vertical position of the actor as a double.

Methods:

1. move(int distance)

- **Purpose**:
  - Overrides Actor's move method to add precision support by converting the integer distance to double and delegating to the overloaded move(double distance) method.
- **Implementation**:
  - Casts the integer distance to a double and calls move(double distance).

2. move(double distance)

- **Purpose**:
  - Moves the actor forward by a precise distance based on its rotation angle.
- **Implementation**:
  - Converts the actor's rotation to radians using Math.toRadians(getRotation()).
  - Calculates changes in x (dx) and y (dy) using trigonometric functions.
  - Updates the exact position and sets the new location using setLocation(double x, double y).

3. setLocation(double x, double y)

- **Purpose**:
  - Sets the actor's position using precise coordinates.
- **Implementation**:
  - Updates exactX and exactY.

- Calls super.setLocation() with rounded integer values to display the actor at the nearest pixel location.

4. setLocation(int x, int y)

- **Purpose**:
  - Overrides Actor's method to synchronize exactX and exactY with integer coordinates.
- **Implementation**:
  - Updates exactX and exactY with the integer values and calls the superclass method.

5. getExactX()

- **Purpose**:
  - Returns the exact horizontal coordinate as a double.
- **Usage**:
  - Useful for collision detection, precise calculations, or debugging.

6. getExactY()

- **Purpose**:
  - Returns the exact vertical coordinate as a double.
- **Usage**:
  - Similar to getExactX(), useful for precision tasks.

VI. Conclusion

Working on the "Plant vs. Zombie" project taught us that a game is much more than just software.

Over two months, we realized that for a game to be enjoyable, it needs rich content, engaging gameplay, and appealing visuals—just like a web server needs quality content to be useful.

While developing the game, we had to consider all aspects: the environment, storyline, gameplay mechanics, artwork, and animations. This project allowed us to apply classroom lessons practically, deal with numerous bugs, and encourage self-directed learning. It reinforced that a computer science major requires continuous self-study, as the IT world is vast and ever-evolving. Our team is committed to completing this game and publishing it as our first project, aiming to develop more applications that enhance the user experience.

And we would like to express our sincere gratitude to our tutor, Mr. Tung, for guiding us through the Object-Oriented Programming (OOP) project. Your support, insightful feedback, and dedication are invaluable. This project has significantly enhanced us in understanding and applicating OOP principles. Thank you for providing an exciting environment of exploration and learning.