Simple Game

Single Player

Simplified Features

Addict Easily

PLANTS.VS ZOMBIES

PROJECT REPORT

DEVELOPER TEAM

Tran Đang Nhat

Le Đoan Cuong Thinh

Đo Phu Thanh

Đang Nguyen Nhat Vy

Phan Van Tai Anh

ADVISOR

Prof. Tran Thanh Tung

The aim of this game is to defend your home from waves of invading zombies by strategically planting different types of plants in your garden. Each plant has unique abilities to combat zombies, and the goal is to prevent the zombies from reaching your house and "eating your brains." The game is won when the last zombie has been defeated. Players can manage the order of plants in the 5x9 garden to maximize the efficiency of plants.

Contents

CHAP 1: INTRODUCTION

1.1 From being simple to complicate

In the process of video game development, it often starts with simple concepts but becomes complex with more ideas which are supported by features of the programming algorithm. From the beginning, a video game was created as a prototype version to test the mechanics, such as the colliding between two or more objects. When the program is continued to develop, more complex features are added like classes, objects, mechanics, friendly interface, and the advanced principle in Oriented-Object Programming to manage the code. Besides that, the aspects such as graphics, animations, sound effects, and user experience add further depth, requiring collaboration across disciplines. This progression from simple mechanics to a polished product reflects the iterative nature of game development, where continuous refinement transforms an idea into an engaging and immersive experience.

By working on a project like Plants vs Zombie as a final project for the "Oriented-Object Programing" subject, we are looking for not only the full-credit for this course but also the opportunity in the improvement of Java developing skills.

1.2 About our game project

The main concept of our game is based on the game named "Plants vs Zombies" released by EA for Android, IOS and PC using Windows OS.

However, due to the limitation of time and development skills, we decided to simplify the size of our game compared to the original game, while maintaining the core concept.

Despite that, we tried to make the game easier to understand, play, and represent the mechanism, to serve the aim of education, and easier enhancing in the future.

In summary, although the product cannot be compared with the original version of Plants vs Zombies, it is still an interesting experiment for players who want to have a short relaxing time.

1.3 The initial concept of the game

As mentioned in point 1.2, we tried to make a simplified version of Plants vs Zombies in Java.
Primarily, we define the rule is to defend against the zombie wave using several types of plants in the garden which has the size of 5x9. Each plant has a unique cost called "Sun point". To earn the "Sun point", players can use the plant which can spawn Sun or collect the random spawn Sun in the garden. The game is won when the last zombie is defeated. Otherwise, the game is lost when a zombie reaches our house.

Some features we have added/ modify:
- Automatic spawn sun and the sun point counter
- Automatic spawn zombies with three types of zombies, including normal zombies, zombie hat, zombie squad corresponding to each the amount of blood
- Three types of main plants with 2 impacts on zombies, including deal damage and slow down zombies, and 1 plant to breed more sun.
- The buying mechanics using sun point.
- Play again if you win the game.

1.4 Reference

- Image from: https://google.com.vn/
- The tutorial Java coding: https://www.youtube.com/@BroCodez/
- The resource: https://github.com/leaser019/

1.5 Developer Team

Name **- Github username**	**UID**	**Contribute**	**%Contribution**
Tran Dang Nhat - *trannhat900*	ITITIU22115	Support every part of the project Sound Slide	98%
Le Doan Cuong Thinh - *Thinkkk*	ITITIU22151	UML design Main Code	100%
Do Phu Thanh - *PThanh04*	ITITWE22159	Support UML design	93%
Dang Nguyen Nhat Vy - *Nhatvy1711*	ITITDK22102	Write report Slide	96%
Phan Van Tai Anh - phananh4848	ITITIU22011	Write report Slide	98%

CHAP 2: SOFTWARE REQUIREMENTS

2.1 What we have

1. User friendly, efficient and lucrative system.
2. Simplified the features from the original game
3. Easy to operate, develop, understand
4. With measured coding and professional thinking.

2.2 Our Objectives

1. Maximum high definition.

2. Design the whole system in an efficient way.

3. Provide updatable list of plants, zombies with the unique abilities

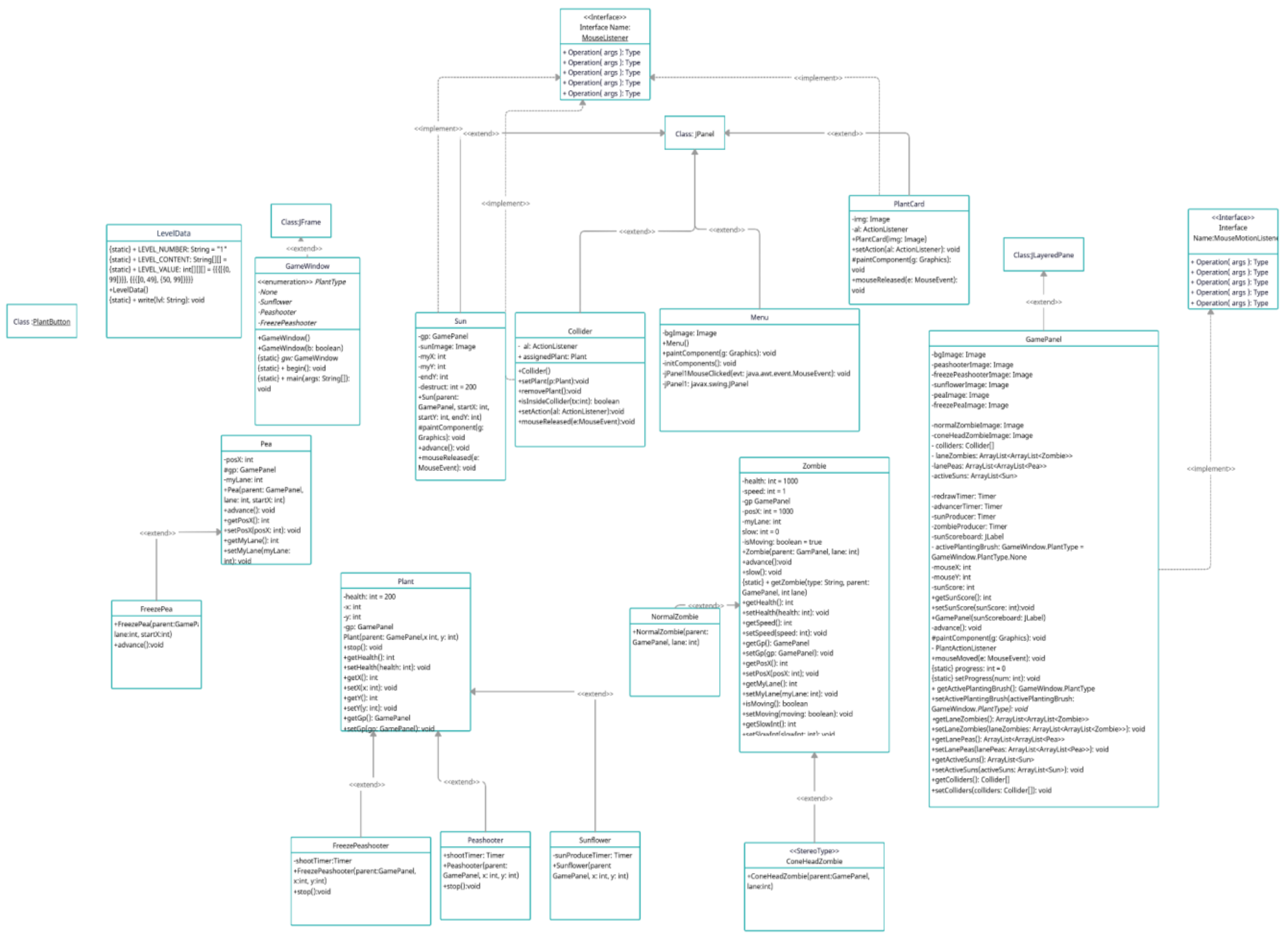
4. Easy to update.

2.3 Working platform

1. Visual Studio Code with Java language pack and additional library

2. Canva/ Photoshop
3. Eclipse with Java language pack and additional library

2.4 Class diagram



<<Interface>>
Interface Name:
MouseListener
+ Operation(args): Type
+ Operation(args): Type
+ Operation(args): Type
+ Operation(args): Type
+ Operation(args): Type
<<implement>>
<<extend>>
<<implement>>
<<implement>>
Class: JPanel
<<extend>>
<<extend>>
<<extend>>
Class:JFrame
<<extend>>
LevelData
{static} + LEVEL_NUMBER: String = "1"
{static} + LEVEL_CONTENT: String[][] =
{static} + LEVEL_VALUE: int[][][] = {{{{0, 99}}}, {{{0, 49}, {50, 99}}}}
+LevelData()
{static} + write(lvl: String): void
Class :PlantButton
GameWindow
<<enumeration>> PlantType
-None
-Sunflower
-Peashooter
-FreezePeashooter
+GameWindow()
+GameWindow(b: boolean)
{static} gw: GameWindow
{static} + begin(): void
{static} + main(args: String[]): void
Sun
-gp: GamePanel
-sunImage: Image
-myX: int
-myY: int
-endY: int
-destruct: int = 200
+Sun(parent: GamePanel, startX: int, startY: int, endY: int)
#paintComponent(g: Graphics): void
+advance(): void
+mouseReleased(e: MouseEvent): void
Collider
- al: ActionListener
+ assignedPlant: Plant
+Collider()
+setPlant(p:Plant):void
+removePlant():void
+isInsideCollider(tx:int): boolean
+setAction(al: ActionListener):void
+mouseReleased(e:MouseEvent):void
Menu
-bgImage: Image
+Menu()
+paintComponent(g: Graphics): void
-initComponents(): void
-jPanel1MouseClicked(evt: java.awt.event.MouseEvent): void
-jPanel1: javax.swing.JPanel
PlantCard
-img: Image
-al: ActionListener
+PlantCard(img: Image)
+setAction(al: ActionListener): void
#paintComponent(g: Graphics): void
+mouseReleased(e: MouseEvent): void
Class:JLayeredPane
<<extend>>
<<Interface>>
Interface
Name:MouseMotionListener
+ Operation(args): Type
+ Operation(args): Type
+ Operation(args): Type
+ Operation(args): Type
+ Operation(args): Type
<<implement>>
GamePanel
-bgImage: Image
-peashooterImage: Image
-freezePeashooterImage: Image
-sunflowerImage: Image
-peaImage: Image
-freezePeaImage: Image
-normalZombieImage: Image
-coneHeadZombieImage: Image
- colliders: Collider[]
- laneZombies: ArrayList<ArrayList<Zombie>>
-lanePeas: ArrayList<ArrayList<Pea>>
-activeSuns: ArrayList<Sun>
-redrawTimer: Timer
-advancerTimer: Timer
-sunProducer: Timer
-zombieProducer: Timer
-sunScoreboard: JLabel
- activePlantingBrush: GameWindow.PlantType = GameWindow.PlantType.None
-mouseX: int
-mouseY: int
-sunScore: int
+getSunScore(): int
+setSunScore(sunScore: int):void
+GamePanel(sunScoreboard: JLabel)
-advance(): void
#paintComponent(g: Graphics): void
- PlantActionListener
+mouseMoved(e: MouseEvent): void
{static} progress: int = 0
{static} setProgress(num: int): void
+ getActivePlantingBrush(): GameWindow.PlantType
+setActivePlantingBrush(activePlantingBrush: GameWindow.PlantType): void
+getLaneZombies(): ArrayList<ArrayList<Zombie>>
+setLaneZombies(laneZombies: ArrayList<ArrayList<Zombie>>): void
+getLanePeas(): ArrayList<ArrayList<Pea>>
+setLanePeas(lanePeas: ArrayList<ArrayList<Pea>>): void
+getActiveSuns(): ArrayList<Sun>
+setActiveSuns(activeSuns: ArrayList<Sun>): void
+getColliders(): Collider[]
+setColliders(colliders: Collider[]): void
Pea
-posX: int
#gp: GamePanel
-myLane: int
+Pea(parent: GamePanel, lane: int, startX: int)
+advance(): void
+getPosX(): int
+setPosX(posX: int): void
+getMyLane(): int
+setMyLane(myLane: int): void
<<extend>>
FreezePea
+FreezePea(parent:GamePanel, lane:int, startX:int)
+advance():void
Plant
-health: int = 200
-x: int
-y: int
-gp: GamePanel
Plant(parent: GamePanel,x int, y: int)
+stop(): void
+getHealth(): int
+setHealth(health: int): void
+getX(): int
+setX(x: int): void
+getY(): int
+setY(y: int): void
+getGp(): GamePanel
+setGp(gp: GamePanel): void
<<extend>>
<<extend>>
<<extend>>
FreezePeashooter
-shootTimer:Timer
+FreezePeashooter(parent:GamePanel, x:int, y:int)
+stop():void
Peashooter
+shootTimer: Timer
+Peashooter(parent: GamePanel, x: int, y: int)
+stop():void
Sunflower
-sunProduceTimer: Timer
+Sunflower(parent GamePanel, x: int, y: int)
Zombie
-health: int = 1000
-speed: int = 1
-gp GamePanel
-posX: int = 1000
-myLane: int
slow: int = 0
-isMoving: boolean = true
+Zombie(parent: GamPanel, lane: int)
+advance():void
+slow(): void
{static} + getZombie(type: String, parent: GamePanel, int lane)
+getHealth(): int
+setHealth(health: int): void
+getSpeed(): int
+setSpeed(speed: int): void
+getGp(): GamePanel
+setGp(gp: GamePanel): void
+getPosX(): int
+setPosX(posX: int): void
+getMyLane(): int
+setMyLane(myLane: int): void
+isMoving(): boolean
+setMoving(moving: boolean): void
+getSlowInt(): int
+setSlowInt(slowInt: int): void
NormalZombie
<<extend>>
+NormalZombie(parent: GamePanel, lane: int)
<<extend>>
<<StereoType>>
ConeHeadZombie
+ConeHeadZombie(parent:GamePanel, lane:int)

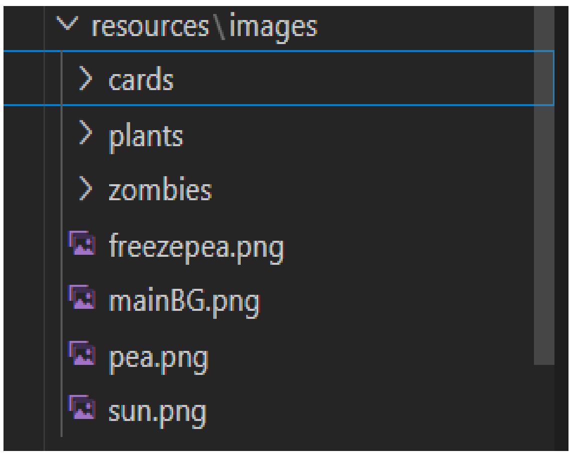
CHAP 3:

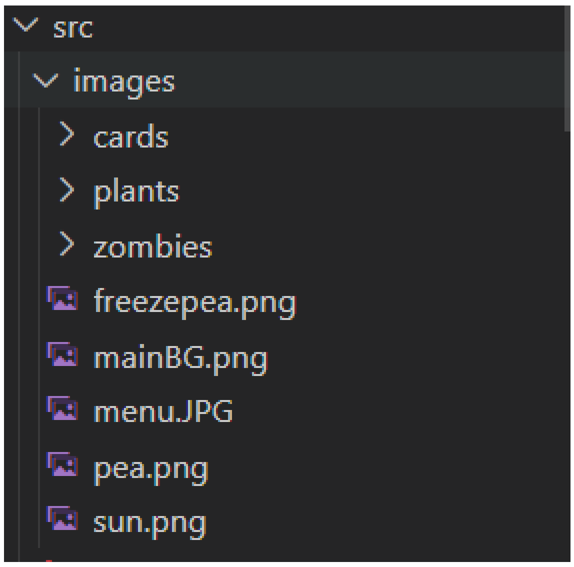
DESIGN. & IMPLEMENTATION

Package Diagram

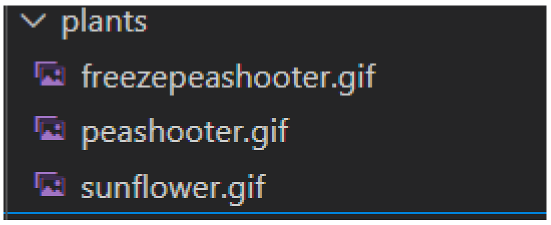
UI Design

We decided to save our resources in a resource folder with some sub folders including plants, cards, and zombies in order to manage them easier. Additionally, we copy them to the src folder to use them in the program:

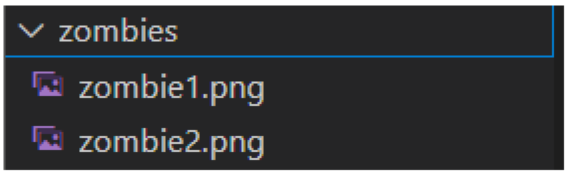

```
resources\images
  cards
  plants
  zombies
  freezepea.png
  mainBG.png
  pea.png
  sun.png
```


```
src
  images
    cards
    plants
    zombies
    freezepea.png
    mainBG.png
    menu.JPG
    pea.png
    sun.png
```

- **Cards:** the folder contains plants cards
- **mainBG,menu:**main background, menu images
- **plants:** Folder contains the gif file (animation) of plants
- **zombies:** Folder contains the png file (animation) of zombies



```
plants
  freezepeashooter.gif
  peashooter.gif
  sunflower.gif
```



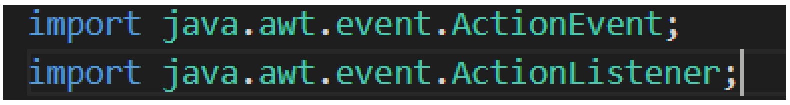
```
zombies
  zombie1.png
  zombie2.png
```

With UI design, we do use the timer, actionevent, and Jpanel in the available Java package, so we can quickly design UI within the code and make pre-implement for the element on the screen.

Java Package

1. **ActionEvent/ActionListener:**

The actionevent/actionlistener class which are available in java.awt.event package represent the event triggered by components, to handle the interaction between user and component such as button. To import to use those classes:



```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

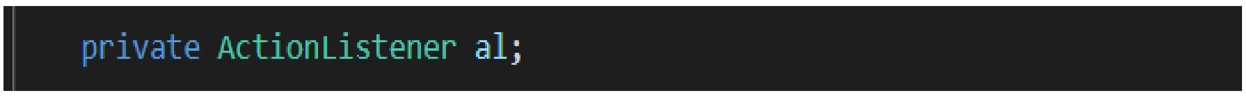
In detail:

ActionListener: defines the behavior of the component when it is interacted.

ActionEvent: provides the event respond the listener, such as animation, sound

To store a new ActionListener object, we implement

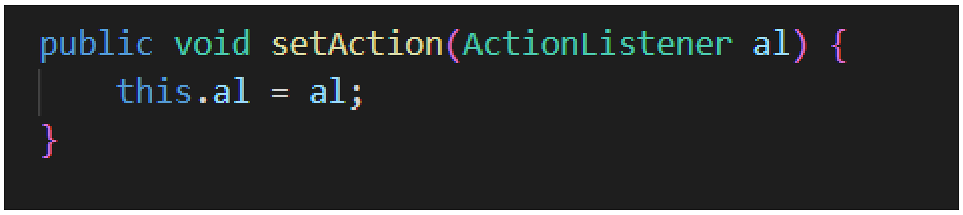
private ActionListener + [name] ;



```java
private ActionListener al;
```

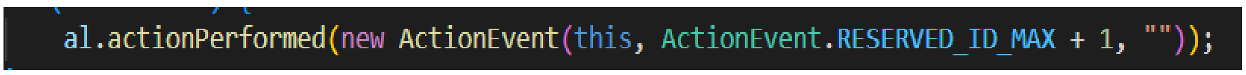
Because we set the object “al” private, we should set a setAction method for this object to make client code be able to use this ActionListener object to perform specific action:

public void setAction(ActionListener + [name])



```java
public void setAction(ActionListener al) {
    this.al = al;
}
```

When an action is defined to happen, the “actionPerfomed” is called:

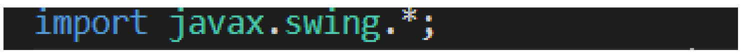


```java
al.actionPerformed(new ActionEvent(this, ActionEvent.RESERVED_ID_MAX + 1, ""));
```

2. java swing:

Swing which is a usable part in Java library, provides high-level GUI components including:
JFrame: The main window of a GUI application.
JPanel: A container for grouping other components.
JLabel: Used to display text or images.
JButton: A button used for actions
ImageIcon: Create the icon [png,gif]
JOptionPane: Displays dialog boxes for messages, confirmations, or input......

To implement the java swing package to import all its classes:



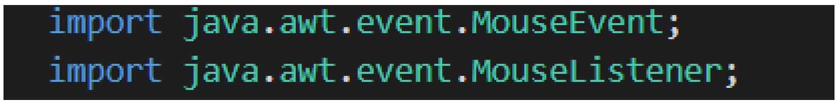
```java
import javax.swing.*;
```

3. MouseEvent/MouseListener

MouseEvent and MouseListener are parts of AWT (Abstracts Window Toolkits) to respond to mouse actions including clicks, movement, and drags. In combination with ActionEvent/ActionListener, we will use to create the action of mouse:

- **void mouseClicked(MouseEvent e)**
Called when the mouse is clicked (pressed and released) on a component.
- **void mousePressed(MouseEvent e)**
Called when a mouse button is pressed down on a component.
- **void mouseReleased(MouseEvent e)**
Called when a pressed mouse button is released on a component.
- **void mouseEntered(MouseEvent e)**
Called when the mouse pointer enters a component.
- **void mouseExited(MouseEvent e)**
Called when the mouse pointer exits a component

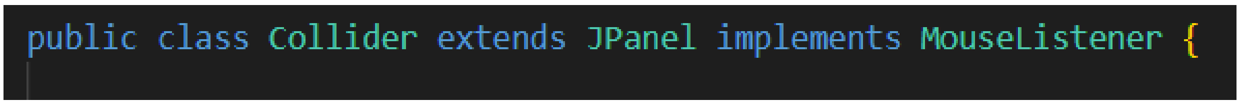
To use those classes in the program:



```
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
```

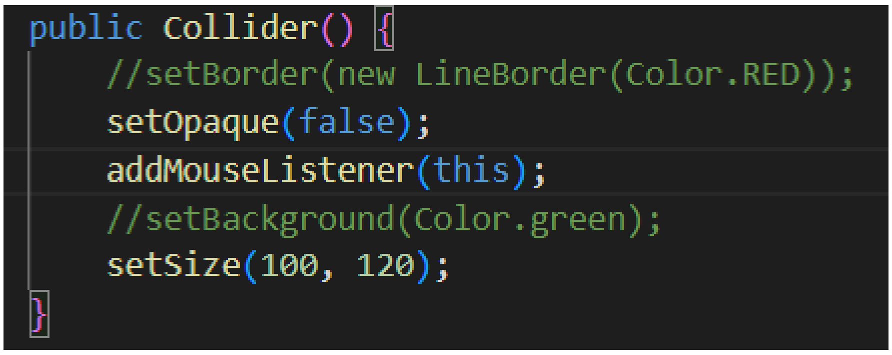
In our project, we apply the Mouse Action classes in several part, such as Collider class:

- implements MouseListener: The class listens for mouse events on itself.



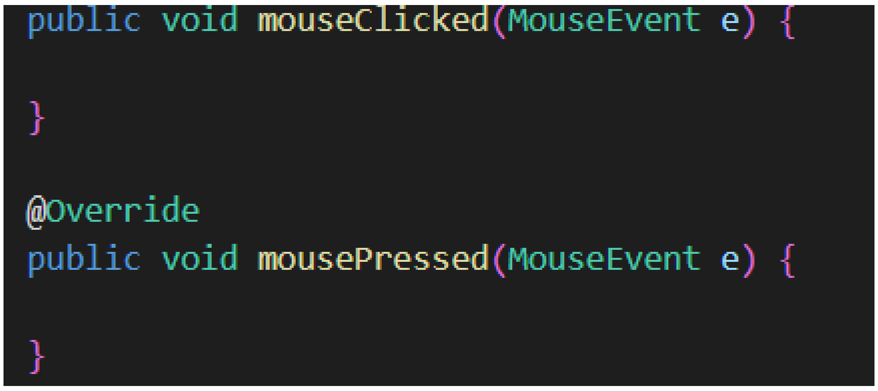
```
public class Collider extends JPanel implements MouseListener {
```

- addMouseListener(this): Registers this Collider instance as the mouse event listener. This means all mouse events on this Collider are handled by its MouseListener methods.



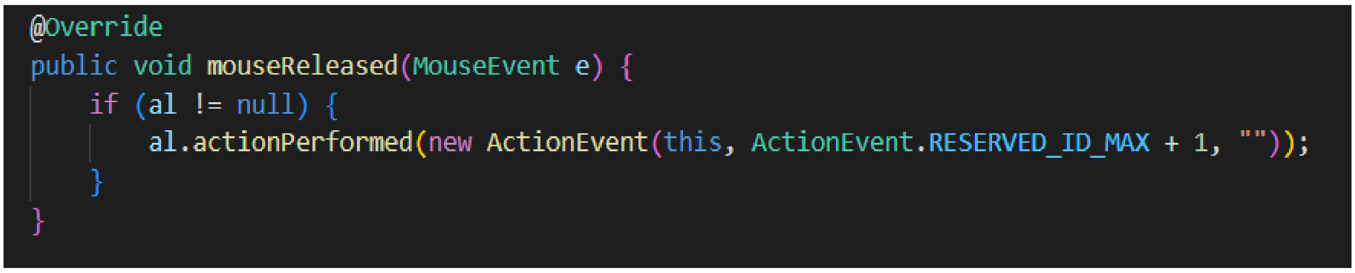
```
public Collider() {
    //setBorder(new LineBorder(Color.RED));
    setOpaque(false);
    addMouseListener(this);
    //setBackground(Color.green);
    setSize(100, 120);
}
```

- **mouseClicked(MouseEvent e):** This method is triggered when the mouse is clicked (pressed and released) on the Collider.
- **mousePress(MouseEvent e):** This method is triggered when the mouse is pressed down on the Collider.



```java
public void mouseClicked(MouseEvent e) {

}

@Override
public void mousePressed(MouseEvent e) {

}
```

- **mouseReleased(MouseEvent e):** This method is triggered when the mouse is released on the Collider.
  - In this method in the Collider classes, the actionPeformed method is triggered by mouseReleased if the ActionListener object named "al" is set

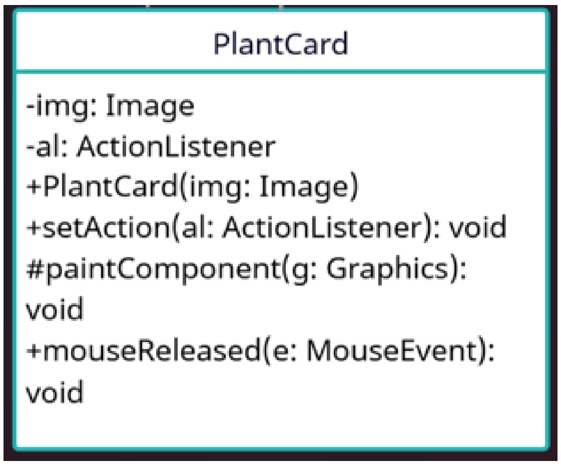


```java
@Override
public void mouseReleased(MouseEvent e) {
    if (al != null) {
        al.actionPerformed(new ActionEvent(this, ActionEvent.RESERVED_ID_MAX + 1, ""));
    }
}
```

4. PlantsCard

This class is built to implement the plant cards, with some features:

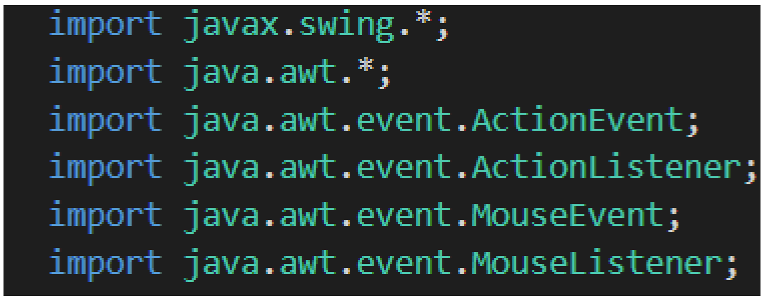
- When the cardCards is drag[Press] and drop[Release], the action is triggered
- Each card is presented by an Image



```
PlantCard
-img: Image
-al: ActionListener
+PlantCard(img: Image)
+setAction(al: ActionListener): void
#paintComponent(g: Graphics):
void
+mouseReleased(e: MouseEvent):
void
```

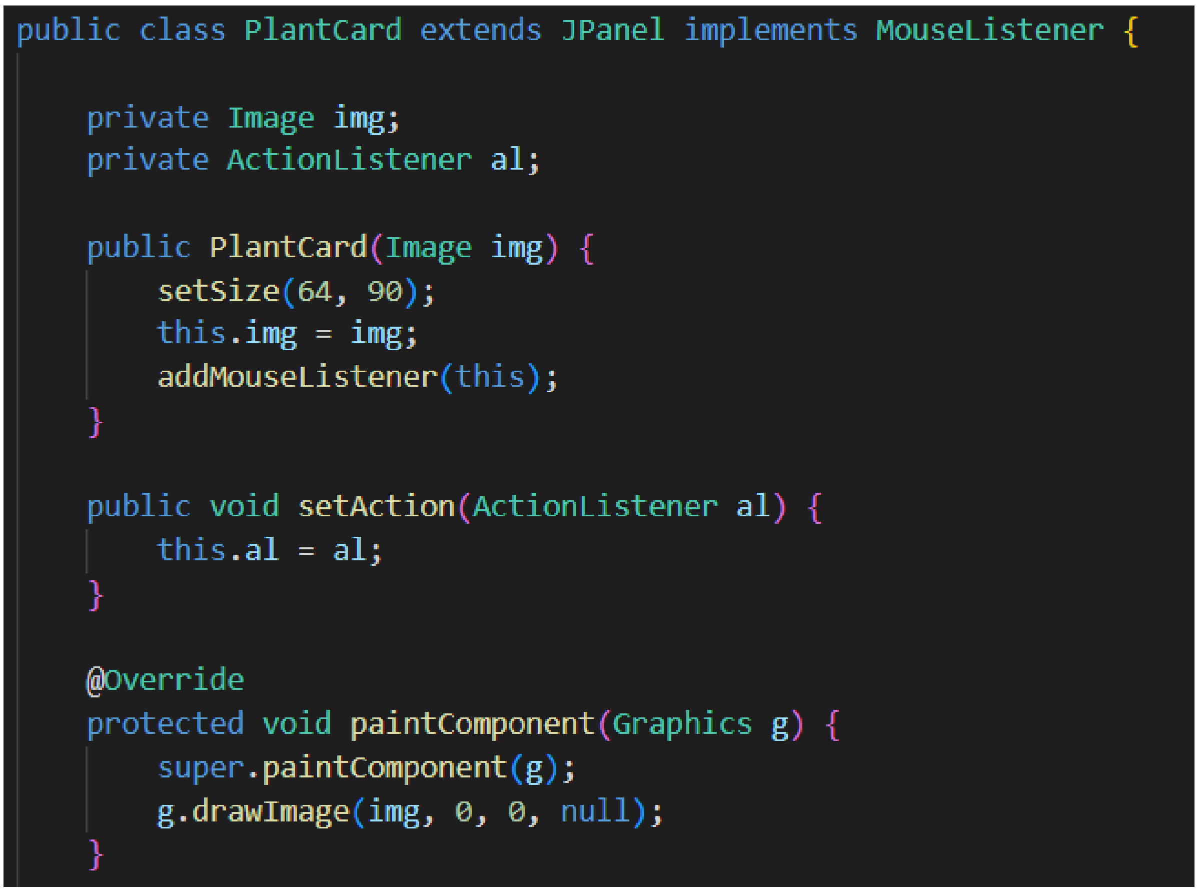
To fulfill those objectives, we apply the ActionEvent, ActionListener to perform the action when the mouse is triggered. MouseEvent/MouseListener to perform the interaction of the player through the mouse. and awt/swing to import the Jpanel, Graphic to serve the graphic purpose.

We import:



```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
```

To declare Image object to store the image of cards besides the ActionListener object:



```java
public class PlantCard extends JPanel implements MouseListener {

    private Image img;
    private ActionListener al;

    public PlantCard(Image img) {
        setSize(64, 90);
        this.img = img;
        addMouseListener(this);
    }

    public void setAction(ActionListener al) {
        this.al = al;
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawImage(img, 0, 0, null);
    }
```

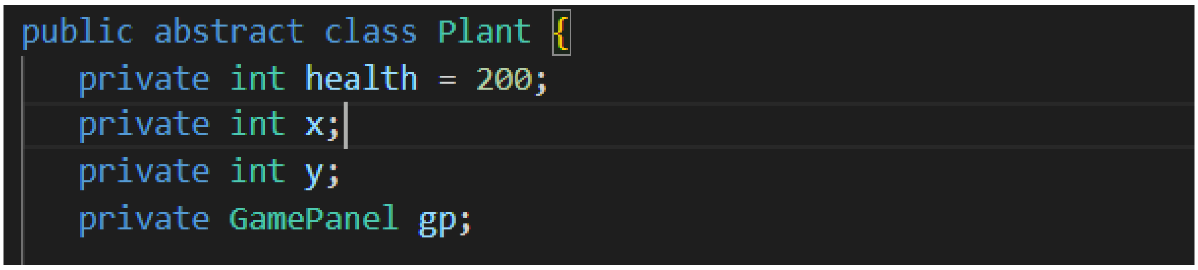
- **setAction method:** Assign a ActionListener to a PlantCard, ready to be called in specific event [Mouse release]
- **paintComponent:** the method which is used to paint a PlantCard, including:

- Background painting ensure: super.paintComponent(g);
- plantcard painting: g.drawImage(img, 0, 0, null);

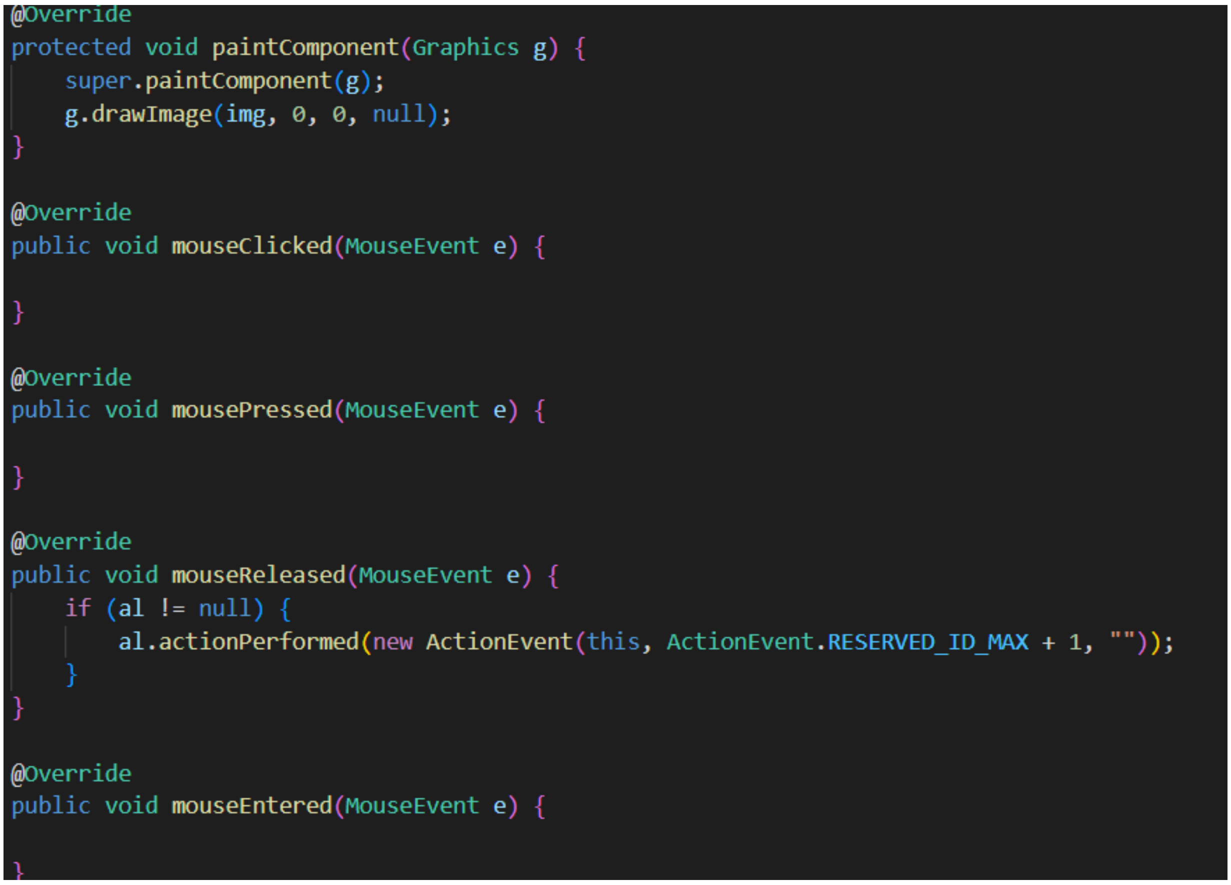
The set of MouseAction/ Listener is built to perform the interaction of user to UI:

5. Plants

We use Plant class as an abstract class that provides a foundation for plant objects, including Peashooter, Sunflower, Freeze peashooter. This class contains the methods to define health, position, and the image of the plant. It is used to be extended to subclasses of specific plants. We define some attributes that Plant class use in its methods:



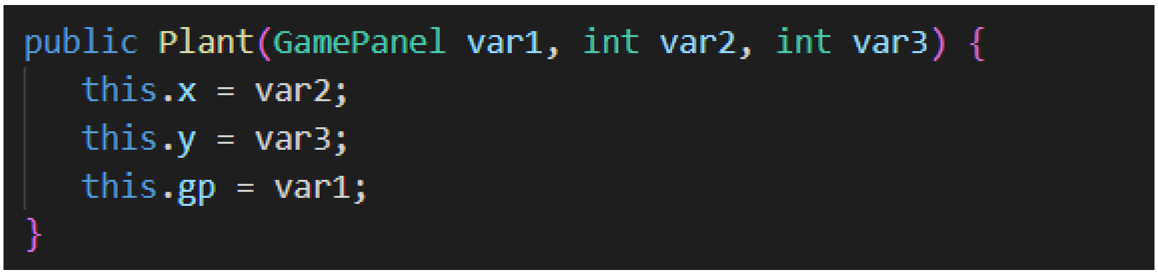
```java
public abstract class Plant {
    private int health = 200;
    private int x;
    private int y;
    private GamePanel gp;
```



```java
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawImage(img, 0, 0, null);
}

@Override
public void mouseClicked(MouseEvent e) {

}

@Override
public void mousePressed(MouseEvent e) {

}

@Override
public void mouseReleased(MouseEvent e) {
    if (al != null) {
        al.actionPerformed(new ActionEvent(this, ActionEvent.RESERVED_ID_MAX + 1, ""));
    }
}

@Override
public void mouseEntered(MouseEvent e) {

}
```

Plant contains four main attributes including: health, x, y, and gp. The properties of attributes are shown below:

- health: An integer initialized to 200, presumably representing the plant's health.
- x and y: Integers representing the coordinates of the plant in the game panel.
- gp: A reference to a GamePanel object, which represents the context or container in which the plant exists.



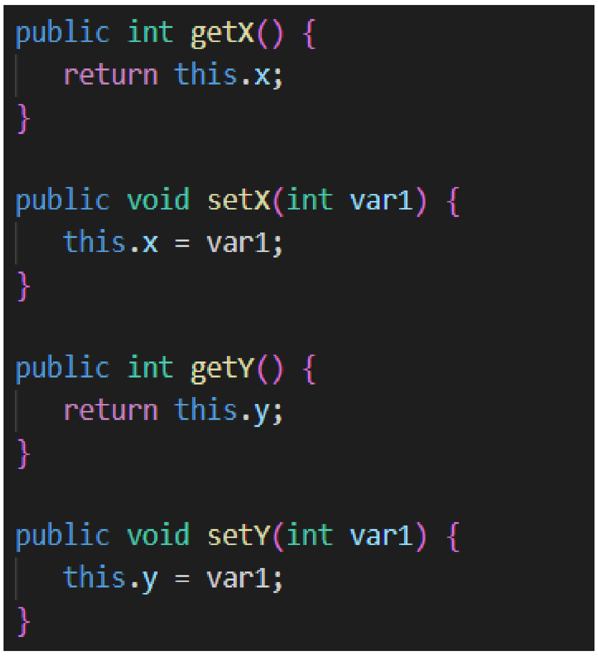
```java
public Plant(GamePanel var1, int var2, int var3) {
   this.x = var2;
   this.y = var3;
   this.gp = var1;
}
```

We implement the constructor:

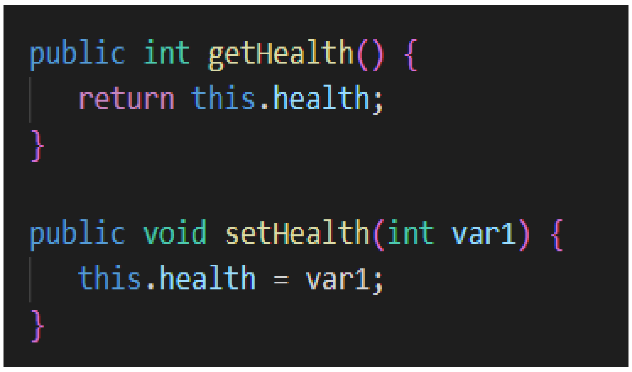
This constructor initializes the x and y position of the plant and the reference to the GamePanel object. We do not take health as a parameter because it has a default value of 200. However, subclasses like Peashooter or Freezepeashooter might allow for different initial health values. This constructor is key because it ensures that every plant object will be placed in a GamePanel and have a specific position on the screen.

Then, We provide several getter and setter methods for all attributes:

- **getHealth()** and **setHealth(int var1)**: we use those to get and set the



```java
public int getX() {
    return this.x;
}

public void setX(int var1) {
    this.x = var1;
}

public int getY() {
    return this.y;
}

public void setY(int var1) {
    this.y = var1;
}
```



```java
public int getHealth() {
    return this.health;
}

public void setHealth(int var1) {
    this.health = var1;
}
```

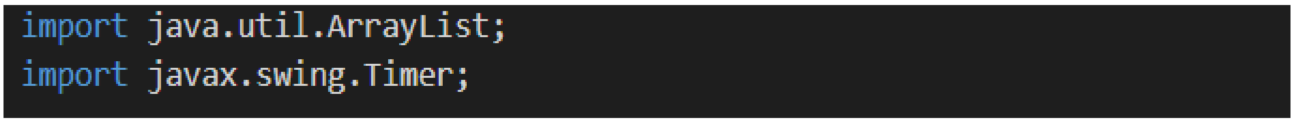
plant's health for managing the plant's condition in the game. In subclasses, we use those encapsulators to update the condition of plants.

- **getX()**, **setX(int var1)**: Get and set the plant's horizontal position.
- **getY()**, **setY(int var1)**: Get and set the plant's vertical position.
- **getGp()**, **setGp(GamePanel var1)**: Get and set the game panel object that this plant belongs to.

There are three classes that extend the Plant class to create specific plants:

a, Peashooter

We construct the Peashooter class as the subclass of Plant to create the Peashooter in the game. This plant will shoot a list of pea automatically if there is a zombie appearing in its lane. Therefore, we use Timer to create the shooting mechanism and Arraylist to store the data of peas. To use the timer class inJava

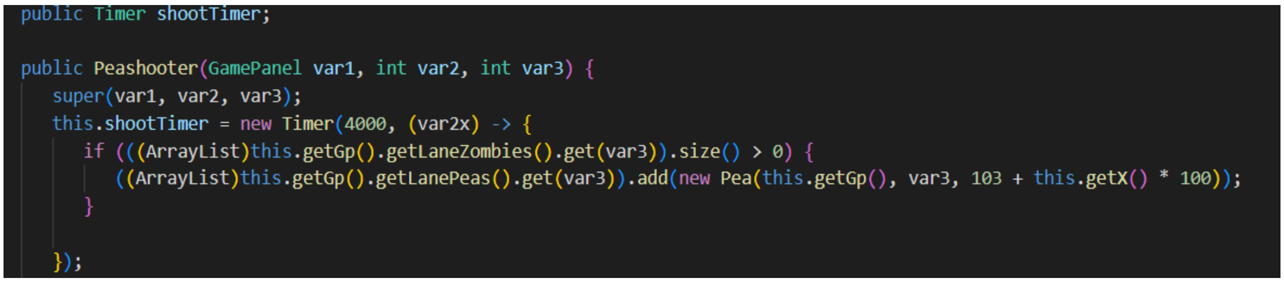


```java
import java.util.ArrayList;
import javax.swing.Timer;
```

swing and Arraylist in this class:

Because Peashooter is the extend class of Plant, we use [super] to call the constructor of Plant to initialize the position and the GamePanel:

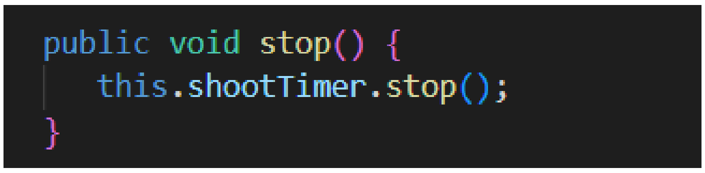
Additionally, we want the peashooter shoot pea every 4 second, so we set:



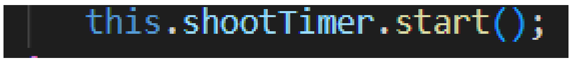
```
public Timer shootTimer;

public Peashooter(GamePanel var1, int var2, int var3) {
    super(var1, var2, var3);
    this.shootTimer = new Timer(4000, (var2x) -> {
        if (((ArrayList)this.getGp().getLaneZombies().get(var3)).size() > 0) {
            ((ArrayList)this.getGp().getLanePeas().get(var3)).add(new Pea(this.getGp(), var3, 103 + this.getX() * 100));
        }

    });
```

this.shootTimer = new Timer(4000, (var2x)

- Inside this lambda, the code checks if there are any zombies in the current lane var3, using the line **((ArrayList)this.getGp().getLaneZombies().get(var3)).**
- If zombies exist in that lane, a new Pea object is added to the lanePea list for that lane **(((ArrayList)this.getGp().getLanePeas().get(var3)).add(...))**. The Pea object represents the projectile that will be shot by the Peashooter.
- The Pea constructor takes GamePanel, the lane index var3, and a calculated x coordinate based on the plant's position **(this.getX() * 100 + 103)**.



```
public void stop() {
    this.shootTimer.stop();
}
```

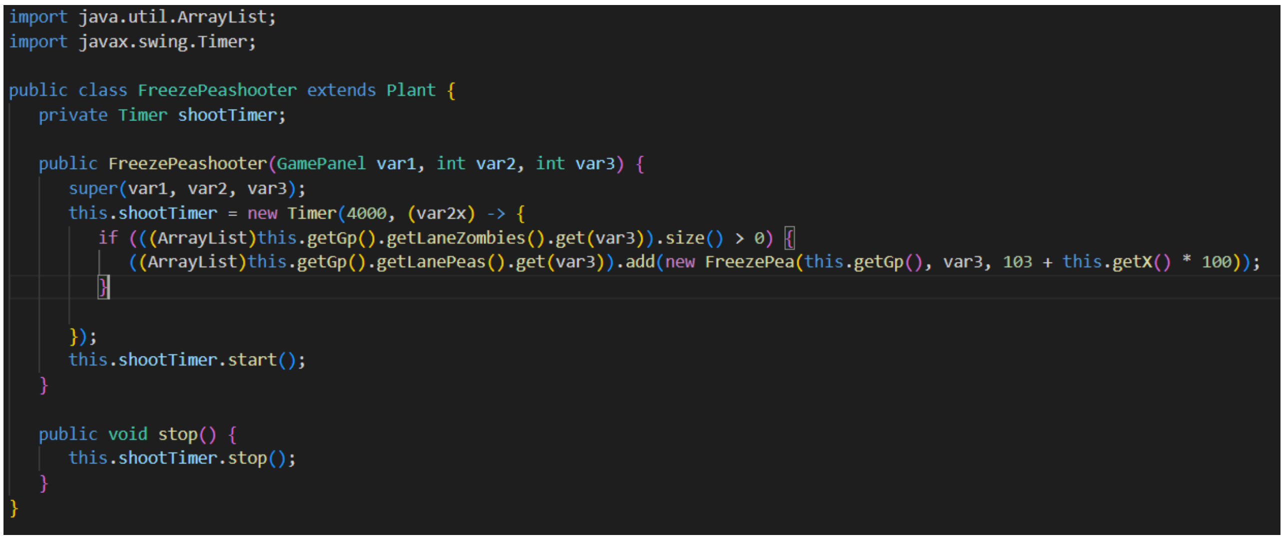


```
    this.shootTimer.start();
```

- We use **stop()** method to stop the Timer in case the Peashooter is no longer active or needs to stop shooting for any reason(The plant die or there are no zombie in its lane)

b, Freezepeashooter

The working of the Freeze peashooter is almost similar to the Peashooter. Therefore, we reuse the code from class Peashooter. There are two things which make Peashooter and Freezepeashooter different:



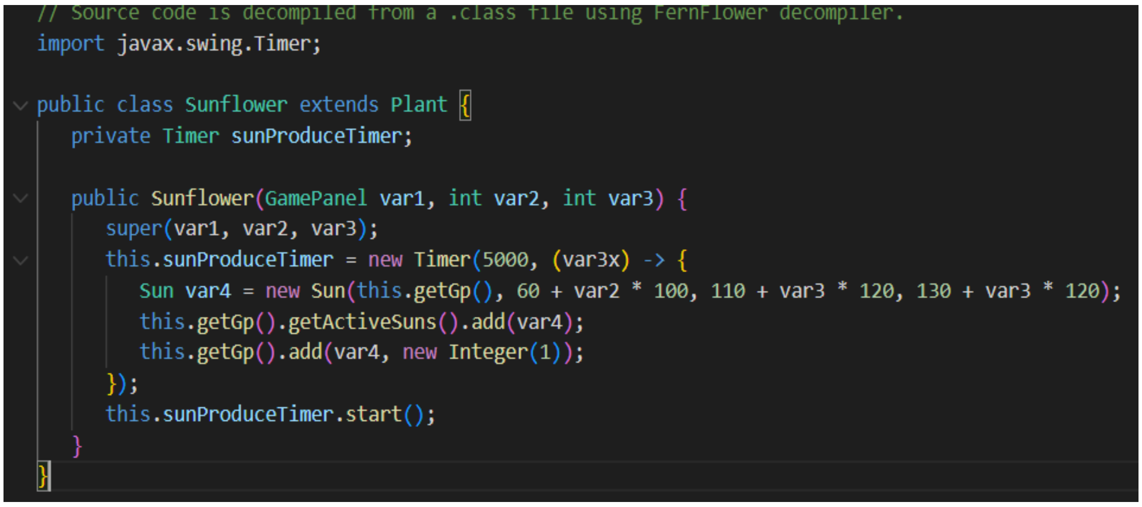
```java
import java.util.ArrayList;
import javax.swing.Timer;

public class FreezePeashooter extends Plant {
   private Timer shootTimer;

   public FreezePeashooter(GamePanel var1, int var2, int var3) {
      super(var1, var2, var3);
      this.shootTimer = new Timer(4000, (var2x) -> {
         if (((ArrayList)this.getGp().getLaneZombies().get(var3)).size() > 0) {
            ((ArrayList)this.getGp().getLanePeas().get(var3)).add(new FreezePea(this.getGp(), var3, 103 + this.getX() * 100));
         }

      });
      this.shootTimer.start();
   }

   public void stop() {
      this.shootTimer.stop();
   }
}
```

- Now, freezepeashooter use the Image of freeze peashooter when it is called in other class [Menu and GameWindow]
- The peashooter shoots pea (call Pea's methods) while Freeze peashooter shoots freeze pea (call FreezePea's method)

c, Sunflower

The final subclass that extends the Plant class is Sunflower. This class provides data of Sun flower which produces Sun(the resource used in the game). Our idea is use the Timer to perform the mechanism of this plant (produce Sun after 7.5 seconds).

- **private Timer sunProduceTimer:** This Timer controls the interval at which



```java
// Source code is decompiled from a .class file using FernFlower decompiler.
import javax.swing.Timer;

public class Sunflower extends Plant {
   private Timer sunProduceTimer;

   public Sunflower(GamePanel var1, int var2, int var3) {
      super(var1, var2, var3);
      this.sunProduceTimer = new Timer(5000, (var3x) -> {
         Sun var4 = new Sun(this.getGp(), 60 + var2 * 100, 110 + var3 * 120, 130 + var3 * 120);
         this.getGp().getActiveSuns().add(var4);
         this.getGp().add(var4, new Integer(1));
      });
      this.sunProduceTimer.start();
   }
}
```

the Sunflower produces sun.

- To initialize the position and the GamePanel, we use **super(var1, var2, var3);** to call the constructor of the Plant class.
- The Timer is set to trigger every 5000 milliseconds (5 seconds). When the timer ticks, the lambda expression is executed, producing a new Sun object.

  this.sunProduceTimer = new Timer(5000, (var3x) -> {...});

- In the lambda expression. A new Sun object is created with the same position as the Sunflower's position.
- The sun object is added to the GamePanel's display and it is ensured that the sun object will be displayed and updated in the game world.
- To start the timer, we implement: **this.sunProduceTimer.start();**

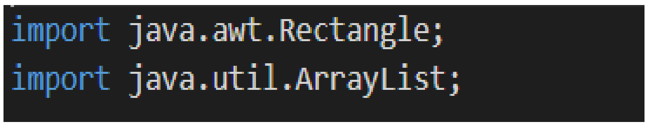
6. Pea

a, Pea

In our game, there are two type of pea(the bullet which is shot by peashooter and freeze peashooter) which are pea and freeze pea. Each type has been created by one class including: class Pea stores data of normal pea, and FreezePea stores the data of freeze pea.

The normal pea is shot by a peashooter in case a zombie exists in its lane, then the pea moves horizontally until it touches the zombie. When the pea collides with a zombie, it deals damage on the zombie while its object data is removed from the game window. So, we create the class Pea which has attributes and methods to simulate the behavior of normal pea when it is called.

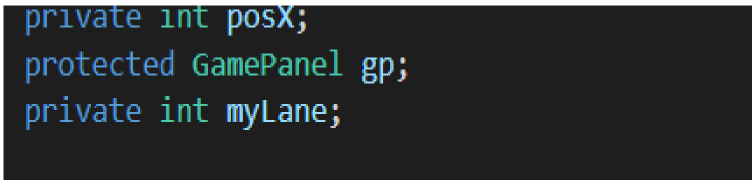
This class requires objects of Rectangle class from awt library and Arraylist class. Those class can be imported by:



```
import java.awt.Rectangle;
import java.util.ArrayList;
```

import java.awt.Rectangle; import java.util.Arraylist;

We need to define the position of peas in their lane (x coordinate), image, and their current lane (y coordinate). So we add three attribute:

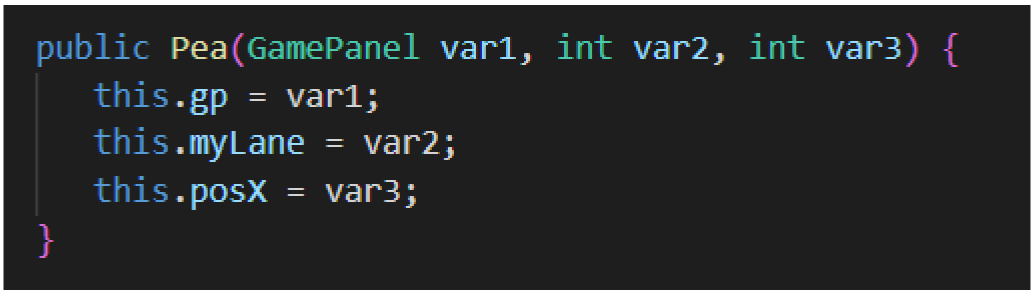


```
private int posX;
protected GamePanel gp;
private int myLane;
```

- **posX:** We use this in methods to track the position of peas, while detecting its collisions with zombie
- **myLane:** to ensure the peas only interact with the zombie in the same lane

- **and gp:** Refers to the GamePanel class, which manages the overall game state.

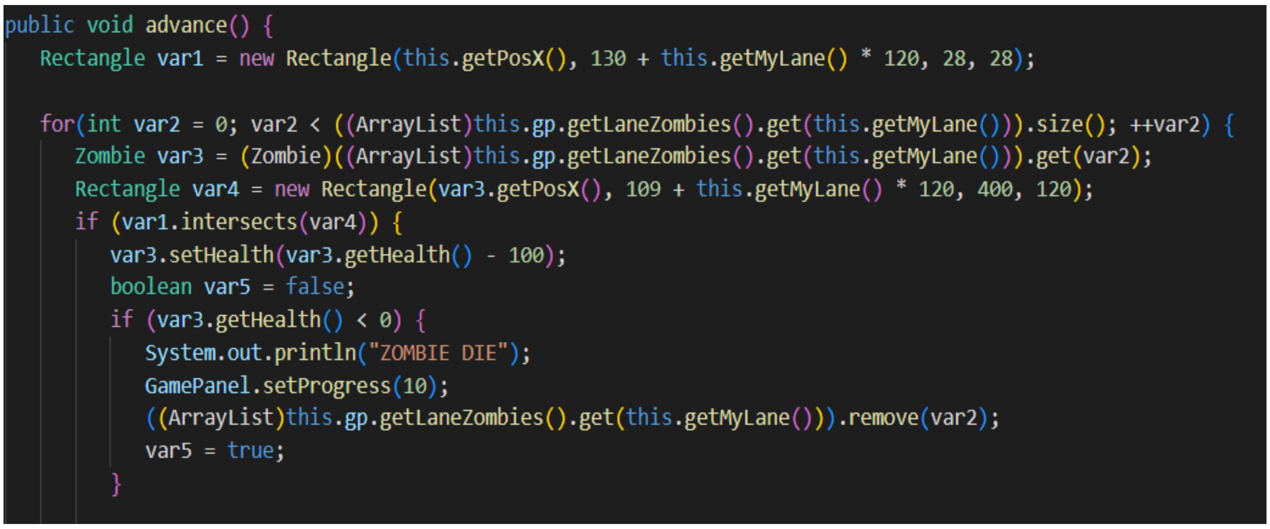
The constructor that we set is for initialize the starting data of Pea (Position, its context in the game panel)



```java
public Pea(GamePanel var1, int var2, int var3) {
    this.gp = var1;
    this.myLane = var2;
    this.posX = var3;
}
```

- var1: Reference to the GamePanel that manages the game.
- var2: The lane where the pea is created.
- var3: The initial horizontal position of the pea.

The objectives of the advance() method handles the pea's primary behavior, which includes movement, collision detection, damage dealing, and removal from the game.



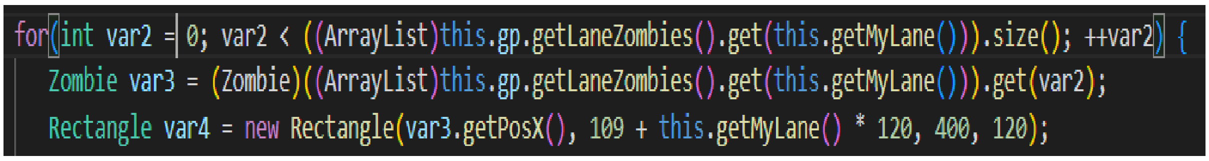
```java
public void advance() {
   Rectangle var1 = new Rectangle(this.getPosX(), 130 + this.getMyLane() * 120, 28, 28);

   for(int var2 = 0; var2 < ((ArrayList)this.gp.getLaneZombies().get(this.getMyLane())).size(); ++var2) {
      Zombie var3 = (Zombie)((ArrayList)this.gp.getLaneZombies().get(this.getMyLane())).get(var2);
      Rectangle var4 = new Rectangle(var3.getPosX(), 109 + this.getMyLane() * 120, 400, 120);
      if (var1.intersects(var4)) {
         var3.setHealth(var3.getHealth() - 100);
         boolean var5 = false;
         if (var3.getHealth() < 0) {
            System.out.println("ZOMBIE DIE");
            GamePanel.setProgress(10);
            ((ArrayList)this.gp.getLaneZombies().get(this.getMyLane())).remove(var2);
            var5 = true;
         }
```

First, the rectangular boundary around the pea (var1) to represent its position and size for collision detection. With

```java
Rectangle var1 = new Rectangle(this.getPosX(), 130 + this.getMyLane() * 120, 28, 28);
```

- **this.getPosX():** retrieves the pea's current horizontal position (posX)

- **130 + this.getMyLane() * 120**: calculates the vertical position of the pea in its lane.
- **this.getMyLane() * 120**: Adjusts the position based on the lane number. Each lane is spaced 120 pixels apart, so this formula shifts the rectangle to the appropriate row.
- **28, 28:** The width and the height of the rectangular that corresponds to the size of the pea.



```
for(int var2 = 0; var2 < ((ArrayList)this.gp.getLaneZombies().get(this.getMyLane())).size(); ++var2) {
   Zombie var3 = (Zombie)((ArrayList)this.gp.getLaneZombies().get(this.getMyLane())).get(var2);
   Rectangle var4 = new Rectangle(var3.getPosX(), 109 + this.getMyLane() * 120, 400, 120);
```

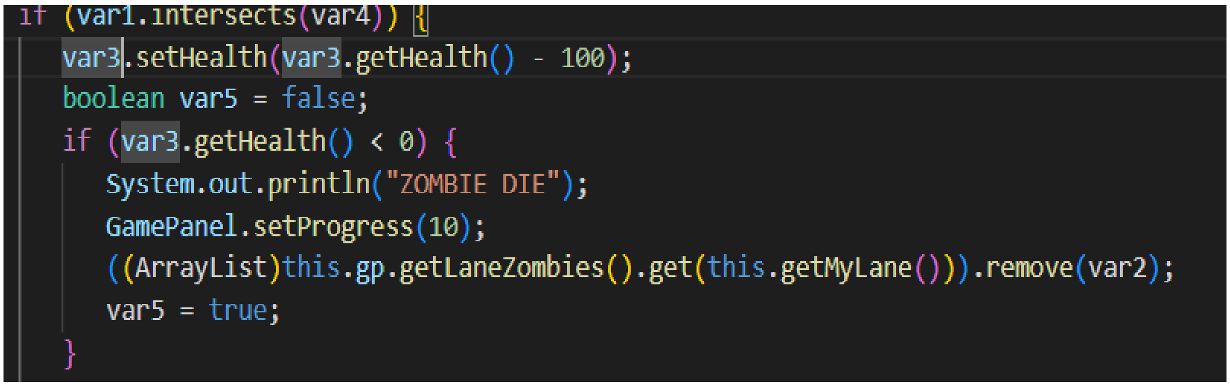
We use loops to create the boundary to check the collision between zombie and pea:

- **this.gp.getLaneZombies().get(this.getMyLane()):** to retrieve the list of zombies in the current lane.
- The loop accesses each zombie in the list using its index var2.
- Each zombie is cast as a Zombie object and stored in var3 for processing.

Create the rectangle for Zombie: **Rectangle var4 =....**

- **var3.getPosX()**: The zombie's horizontal position.
- **109 + this.getMyLane() * 120**: The zombie's vertical position, with 109 is the base vertical offset for zombies.
- **this.getMyLane() * 120** adjusts the position for the specific lane.
- **400**: The width of the rectangle, which accounts for the zombie's potential movement area.
- **120**: The height of the rectangle, matching the zombie's lane height.

In the **if** statement:

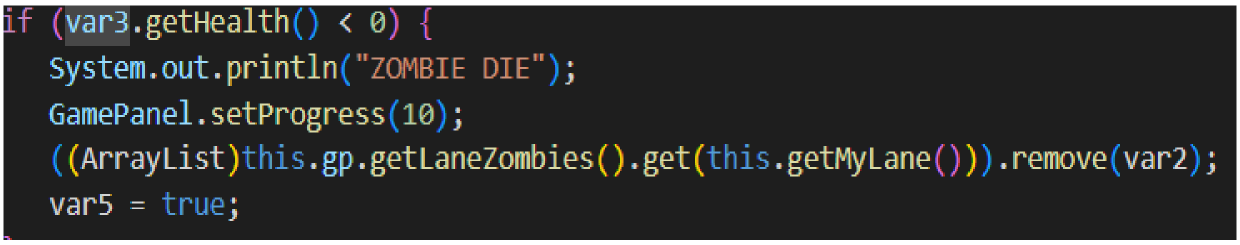


```
if (var1.intersects(var4)) {
   var3.setHealth(var3.getHealth() - 100);
   boolean var5 = false;
   if (var3.getHealth() < 0) {
      System.out.println("ZOMBIE DIE");
      GamePanel.setProgress(10);
      ((ArrayList)this.gp.getLaneZombies().get(this.getMyLane())).remove(var2);
      var5 = true;
   }
```

We use this to check the collision between the pea and the zombie and the action in case there is a collision.

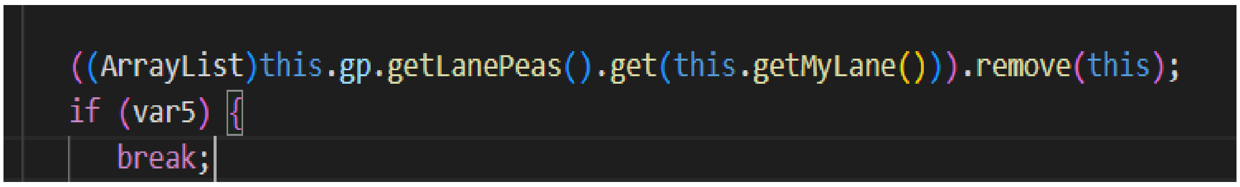
In case of collision:

- **var3.setHealth(var3.getHealth() -100);** update the health of the zombie (health -100). In other words, the damage of a pea is 100.



```java
if (var3.getHealth() < 0) {
   System.out.println("ZOMBIE DIE");
   GamePanel.setProgress(10);
   ((ArrayList)this.gp.getLaneZombies().get(this.getMyLane())).remove(var2);
   var5 = true;
```

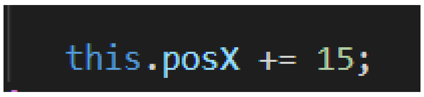
- If the health of zombie becomes zero (0), the system prints the line "ZOMBIE DIE" and removes the zombie from the game panel, besides increasing the Game process by 10 (If the game process is full, we win!).



```java
   ((ArrayList)this.gp.getLanePeas().get(this.getMyLane())).remove(this);
   if (var5) {
      break;
```

- Then **remove(this)** is used to remove the pea from the game panel, means it is no longer active.

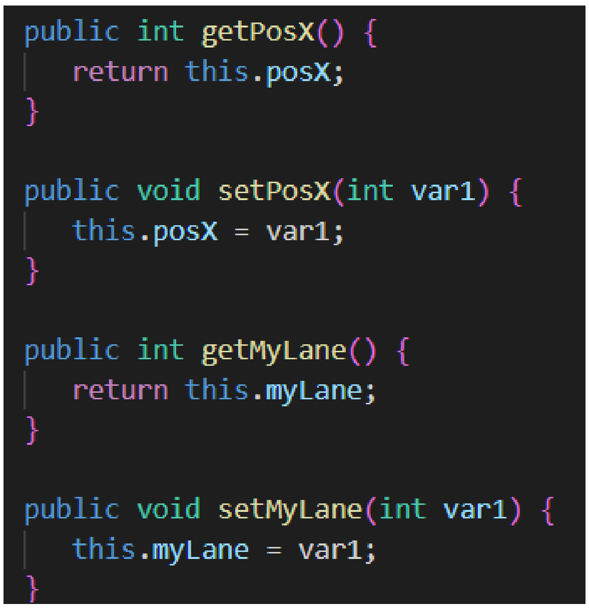
If there is no collision, the pea continues to move from left to right in its lane with the speed of 15.



```java
   this.posX += 15;
```

Additionally, we created the getter and setter of posX and myLane to allow other classes to access the data of pea, such as Peashooter class.

- **setPosX, getPosX:** allow to access and change the data of X coordinate of pea
- **setMyLane, getMyLane:** allow to access and change the data of Y coordinate of pea(its current lane)

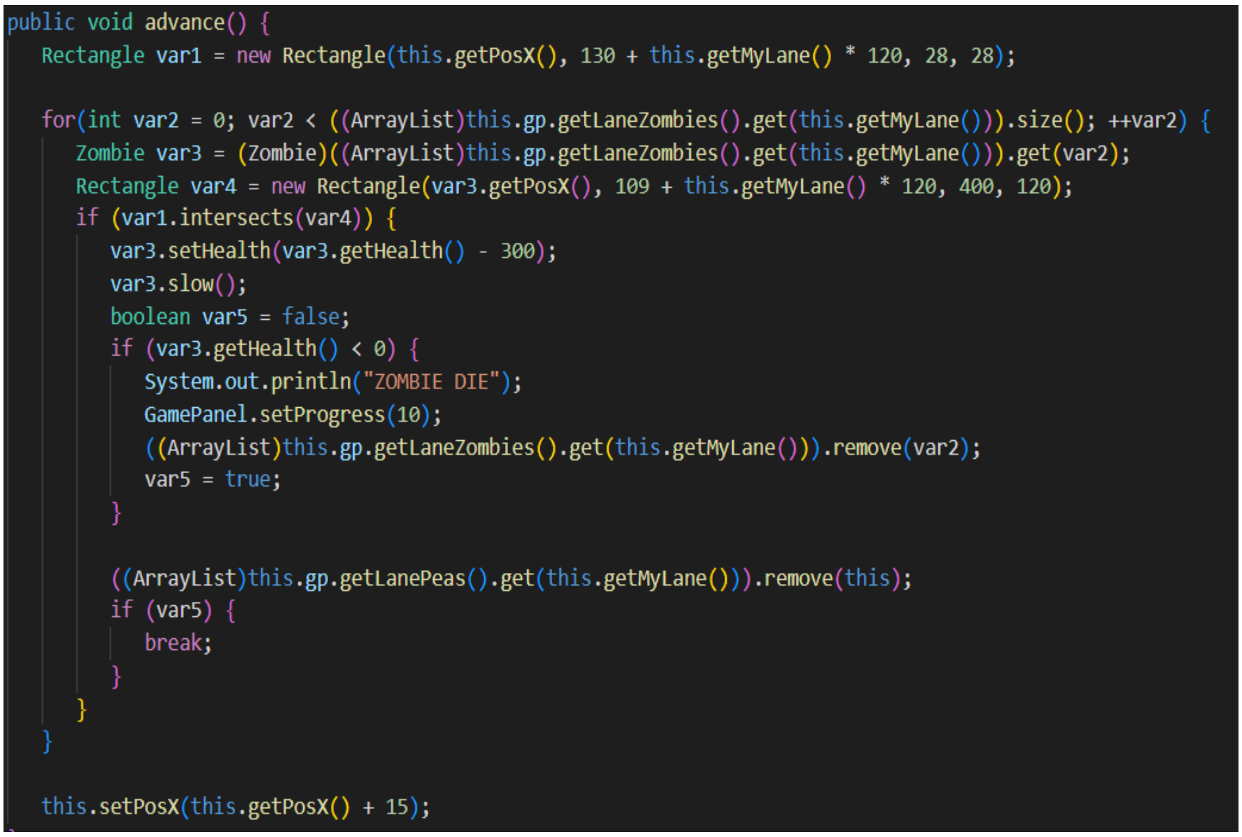


```java
public int getPosX() {
   return this.posX;
}

public void setPosX(int var1) {
   this.posX = var1;
}

public int getMyLane() {
   return this.myLane;
}

public void setMyLane(int var1) {
   this.myLane = var1;
}
```

b, FreezePea

The freeze pea is shot by a freeze peashooter in case a zombie exists in its lane, then the pea moves horizontally until it touches the zombie. When the pea collides with a zombie, it deals damage and “Slow” effect on the zombie while its object data is removed from the game window. So, we created the class FreezePea which has attributes and methods to simulate the behavior of freeze pea when it is called.

This Class is almost similar to the Pea class, except the calling of the method **var3.slow():** represents the slow effect. Therefore, we reuse the code from class Pea to this class.

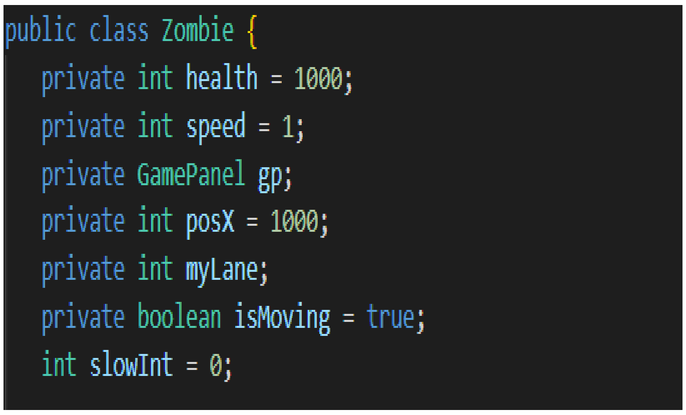


```
public void advance() {
   Rectangle var1 = new Rectangle(this.getPosX(), 130 + this.getMyLane() * 120, 28, 28);

   for(int var2 = 0; var2 < ((ArrayList)this.gp.getLaneZombies().get(this.getMyLane())).size(); ++var2) {
      Zombie var3 = (Zombie)((ArrayList)this.gp.getLaneZombies().get(this.getMyLane())).get(var2);
      Rectangle var4 = new Rectangle(var3.getPosX(), 109 + this.getMyLane() * 120, 400, 120);
      if (var1.intersects(var4)) {
         var3.setHealth(var3.getHealth() - 300);
         var3.slow();
         boolean var5 = false;
         if (var3.getHealth() < 0) {
            System.out.println("ZOMBIE DIE");
            GamePanel.setProgress(10);
            ((ArrayList)this.gp.getLaneZombies().get(this.getMyLane())).remove(var2);
            var5 = true;
         }

         ((ArrayList)this.gp.getLanePeas().get(this.getMyLane())).remove(this);
         if (var5) {
            break;
         }
      }
   }

   this.setPosX(this.getPosX() + 15);
```

7. Zombie

a, Zombie

Zombie is the entity which we should prevent in the game. Our idea is to create different types of zombies with different properties including: health, image... but they have some similar things such as: moving with the same speed until reaching the player's house, only move horizontally in one lane, stop moving when the game is stopped [Win case or lose case]. So we planned to build the main class zombie containing attributes and methods to be used in different extended classes.

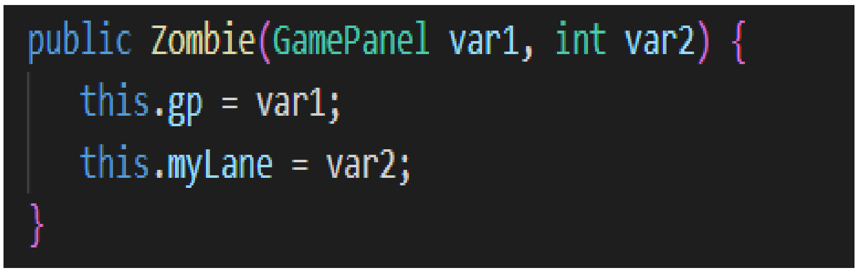


```java
public class Zombie {
    private int health = 1000;
    private int speed = 1;
    private GamePanel gp;
    private int posX = 1000;
    private int myLane;
    private boolean isMoving = true;
    int slowInt = 0;
```

First, we set seven attributes to use in class's method, they are:

- **health**: the health of the zombie, it is set by 1000
- **speed**: the moving speed is set by 0.75
- **gp**: gives the Zombie access to the game's zombies in all lanes.
- posX: the X coordinate of the zombie in the game panel
- myLane: the Y coordinate of zombie in the game panel
- isMoving: this attribute to control the moving state of the zombie.
- slowInt: A counter that slows the zombie's movement when it is affected by certain conditions (like a slowing plant).

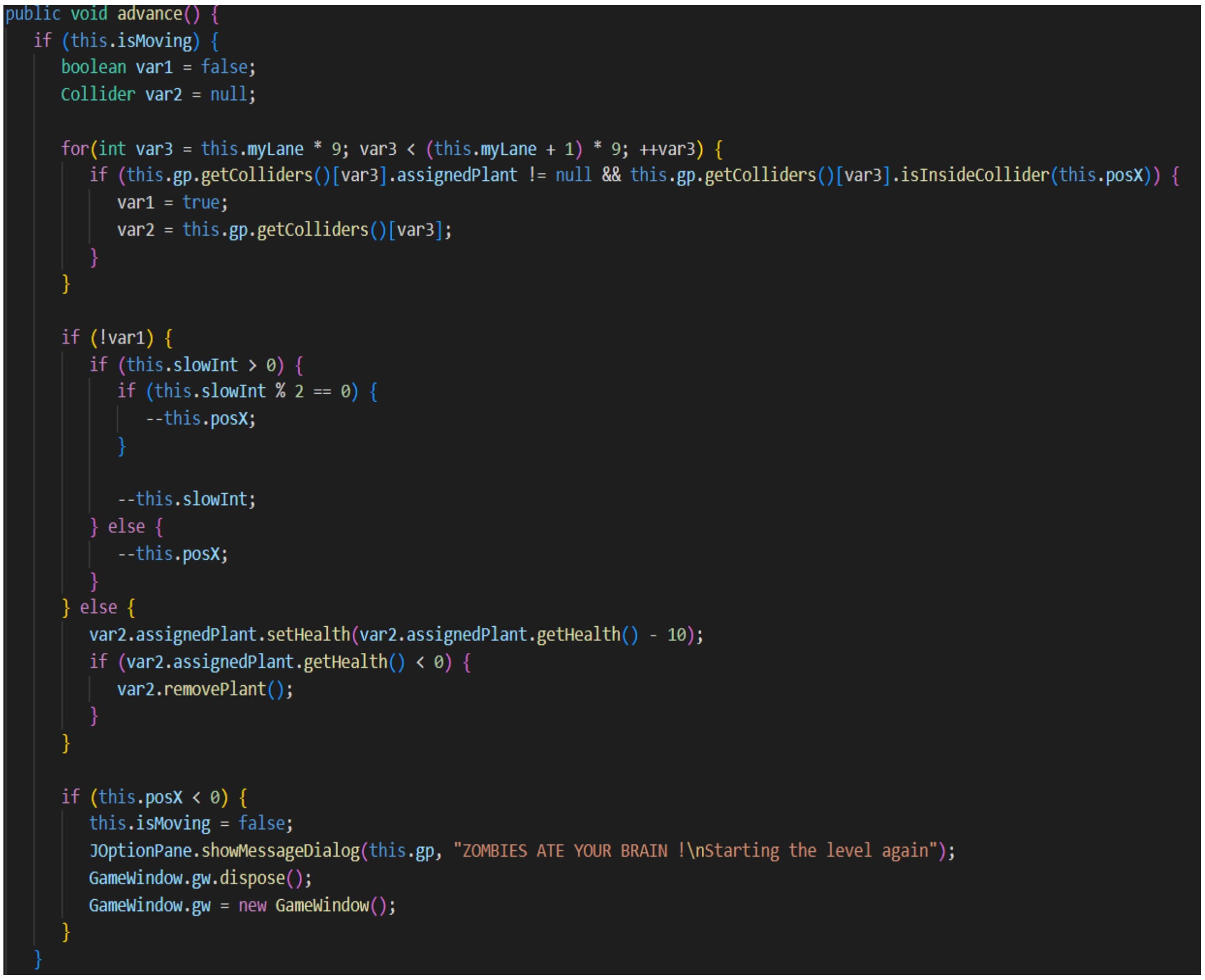
Because we set the values of **health, speed, posX, isMoving, and slowInt,** we just initial **gp, and myLane** in the constructor Zombie:



```java
public Zombie(GamePanel var1, int var2) {
    this.gp = var1;
    this.myLane = var2;
}
```

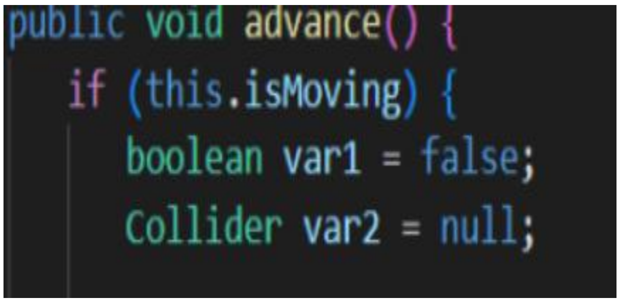
- this.gp = var1; initial value that references the GamePanel that manages the game.
- this.myLane = var2; initial the value that represents the Y coordinate of zombie (the current lane).

Similar to the Pea or Peashooter class, the Zombie's advance method contains features that represent the mechanism of a zombie such as interaction with plants, moving state, health updating, losing case. Specifically:


```java
public void advance() {
    if (this.isMoving) {
        boolean var1 = false;
        Collider var2 = null;

        for(int var3 = this.myLane * 9; var3 < (this.myLane + 1) * 9; ++var3) {
            if (this.gp.getColliders()[var3].assignedPlant != null && this.gp.getColliders()[var3].isInsideCollider(this.posX)) {
                var1 = true;
                var2 = this.gp.getColliders()[var3];
            }
        }

        if (!var1) {
            if (this.slowInt > 0) {
                if (this.slowInt % 2 == 0) {
                    --this.posX;
                }

                --this.slowInt;
            } else {
                --this.posX;
            }
        } else {
            var2.assignedPlant.setHealth(var2.assignedPlant.getHealth() - 10);
            if (var2.assignedPlant.getHealth() < 0) {
                var2.removePlant();
            }
        }

        if (this.posX < 0) {
            this.isMoving = false;
            JOptionPane.showMessageDialog(this.gp, "ZOMBIES ATE YOUR BRAIN !\nStarting the level again");
            GameWindow.gw.dispose();
            GameWindow.gw = new GameWindow();
        }
    }
```

- **In case the zombie is moving [if(this.isMoving)] :**

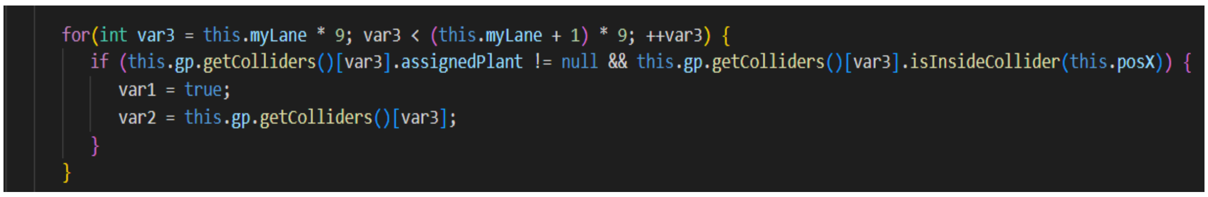
- If **`isMoving`** is `true`, the zombie is allowed to take actions,
- If **`isMoving`** is `false`, the zombie remains stationary.


```java
public void advance() {
    if (this.isMoving) {
        boolean var1 = false;
        Collider var2 = null;
```

- **`boolean var1 = false;`**: **`(var1)`** acts as a flag to indicate whether there is a plant blocking the zombie's path in its lane. It is initially set to `false`, meaning no plant is detected.
- **`Collider var2 = null;`**: **`(var2)`** holds a reference to the specific collider that contains the plant blocking the zombie's path. It is initially set to `null`, it will later be assigned the collider object of the blocking plant if one is found.

- **Check for Plants in the Same Lane:**

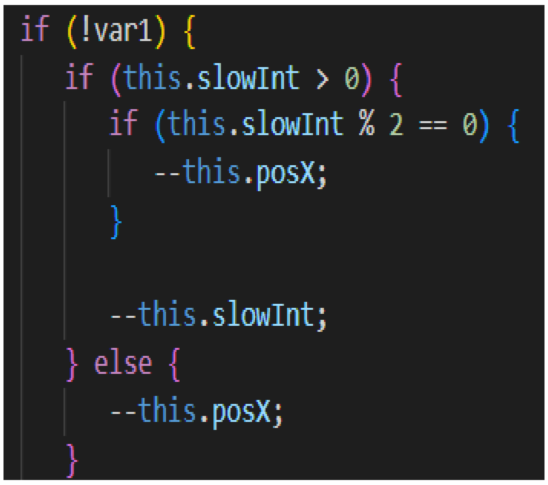
- Each lane has 9 colliders (grid cells or tiles).
- If a plant exists in the collider and the zombie's position (posX) overlaps with it, the zombie interacts with the plant.
- Sets var1 to true and stores the collider (var2) containing the plant.



```java
for(int var3 = this.myLane * 9; var3 < (this.myLane + 1) * 9; ++var3) {
   if (this.gp.getColliders()[var3].assignedPlant != null && this.gp.getColliders()[var3].isInsideCollider(this.posX)) {
      var1 = true;
      var2 = this.gp.getColliders()[var3];
   }
}
```

- **If No Plant is Blocking the Path (!var1):**

- The zombie moves leftward by 1 in each loop (--**posX**).
- If **slowInt** > **0**, it moves slower (every other tick). The counter decrements each time.

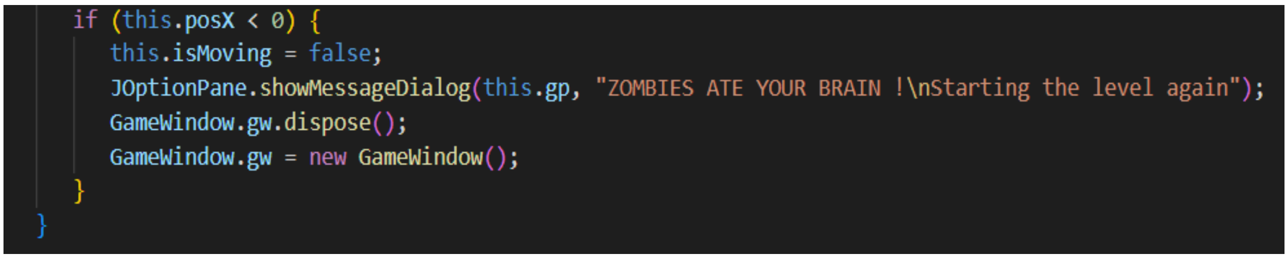


```java
if (!var1) {
   if (this.slowInt > 0) {
      if (this.slowInt % 2 == 0) {
         --this.posX;
      }

      --this.slowInt;
   } else {
      --this.posX;
   }
```

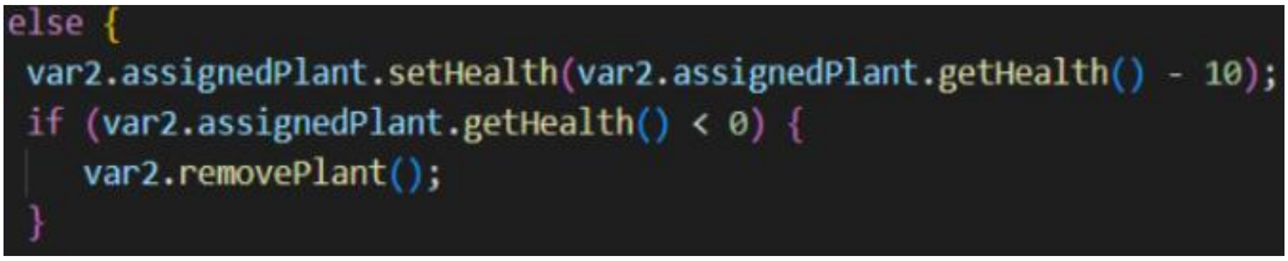
- **If a Plant is Blocking the Path (var1):**

- The zombie attacks the plant, reducing its health by 10 **[getHealth() -10]**
- If the plant's health drops below 0, it is removed **[var2.removePlant();]**

- **LOSING CASE**

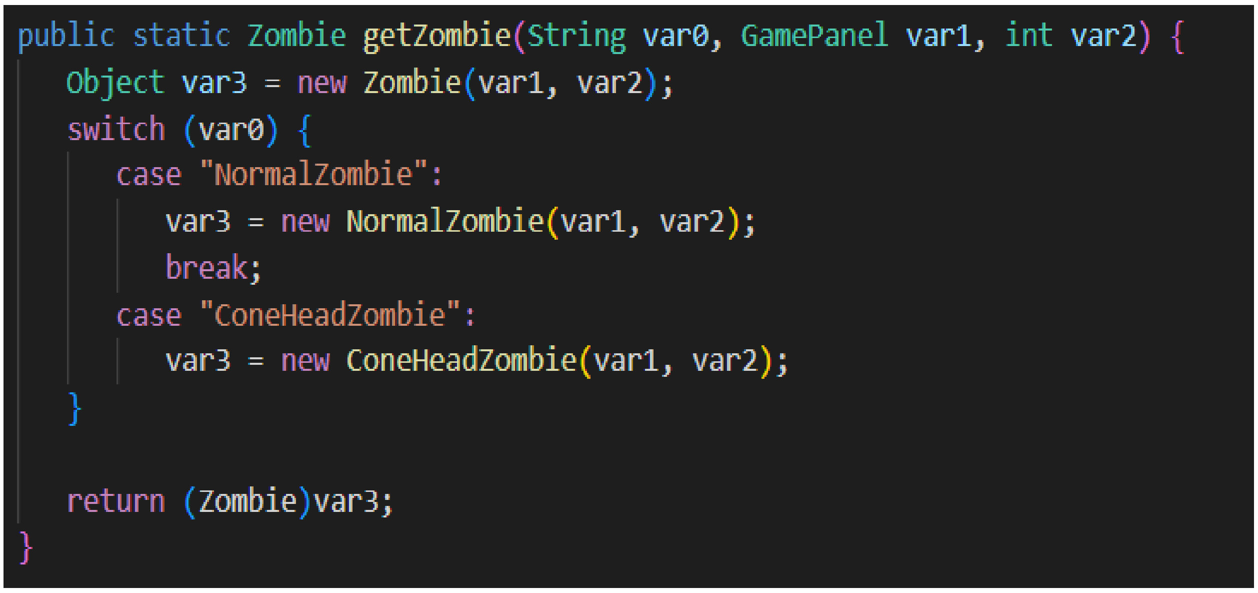


```
    if (this.posX < 0) {
        this.isMoving = false;
        JOptionPane.showMessageDialog(this.gp, "ZOMBIES ATE YOUR BRAIN !\nStarting the level again");
        GameWindow.gw.dispose();
        GameWindow.gw = new GameWindow();
    }
}
```



```
else {
 var2.assignedPlant.setHealth(var2.assignedPlant.getHealth() - 10);
 if (var2.assignedPlant.getHealth() < 0) {
    var2.removePlant();
 }
```

- In case the zombie reaches player’s house (X coordinate become zero)
  if (this.posX <0)
- the zombie stop moving
  this.isMoving = false;
- Then, the system prints out the message “ZOMBIES ATE YOUR BRAIN. Starting the level again” on a message panel.
  JOptionPane.showMessageDialog(this.gp, “ZOMBIES ATE YOUR BRAIN!\nStaring the level again”):
- Finally, the **dispose()** and **GameWindow.gw = new GameWindow()** remove the current window and start a new game window (to reset the game when player loses)



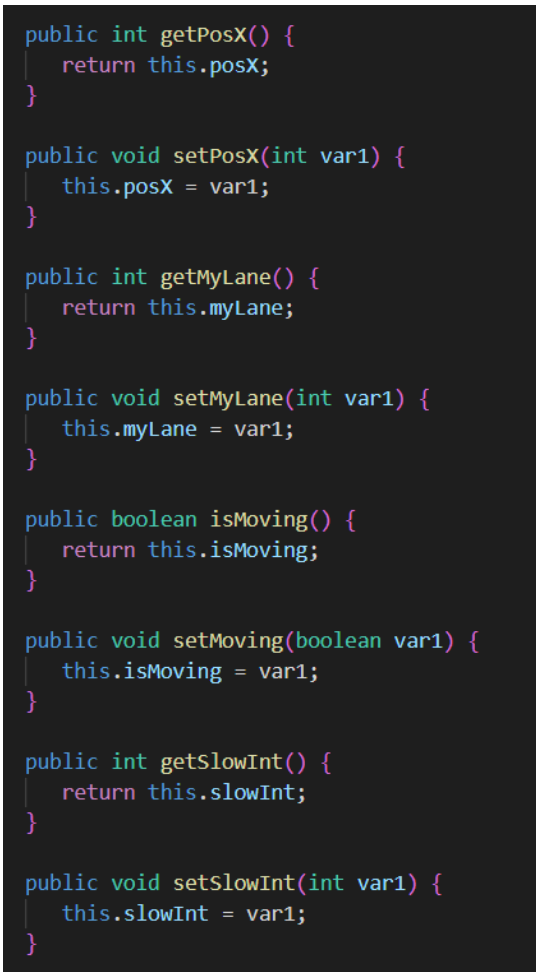
```
public static Zombie getZombie(String var0, GamePanel var1, int var2) {
   Object var3 = new Zombie(var1, var2);
   switch (var0) {
      case "NormalZombie":
         var3 = new NormalZombie(var1, var2);
         break;
      case "ConeHeadZombie":
         var3 = new ConeHeadZombie(var1, var2);
   }

   return (Zombie)var3;
}
```

We use switch statement to implement the cases of the extend class which uses the methods and attributes of Zombie to create different types of zombie:

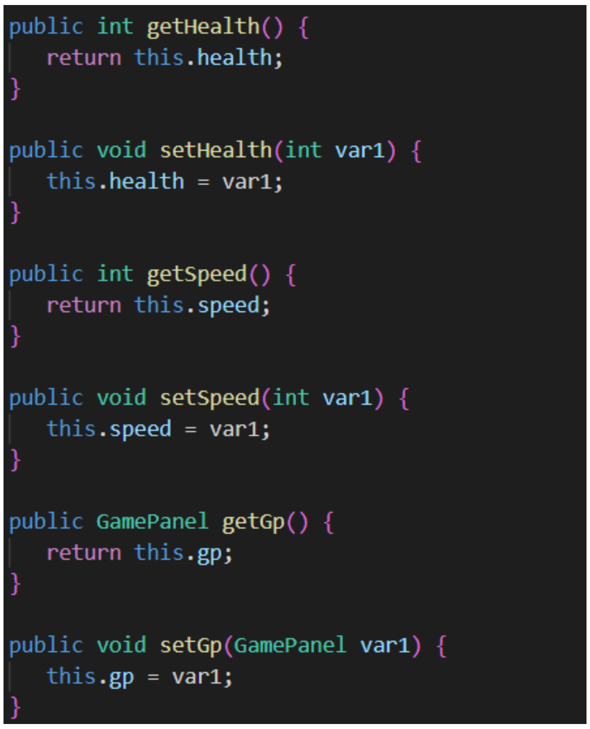
- In the process of creating zombie object: **Object var3 = new Zombie(var1,var2); the switch (var0) is used to choose the type of zombie**
- **case “NormalZombie”:** call the subclass NormalZombie to create a normal zombie.
- **case “ConeHeadZombie”:** call the subclass ConeHeadZombie to create a cone head zombie.

Finally, the set of encapsulators allow the local methods and other classes to access to the attributes which define the state of zombie, including:

- **getHealth/setHealth:** allow to access to the health of zombie
- **getSpeed/setSpee:** allow to access to the speed of zombie
- **getGp/setGp:** allow to access to the game’s zombie in all lanes
- **getposX/setposX:** allow to access to the X coordinate of zombie
- **getLyLane/setMyLane:** allow to access to the Y coordinate of zombie (the current lane)
- **is Moving/setMoving:** allow to access to the moving state of zombie (True = moving, false = not moving)
- **getSlowInt/setSlowInt:** allow to access to the slow state of zombie



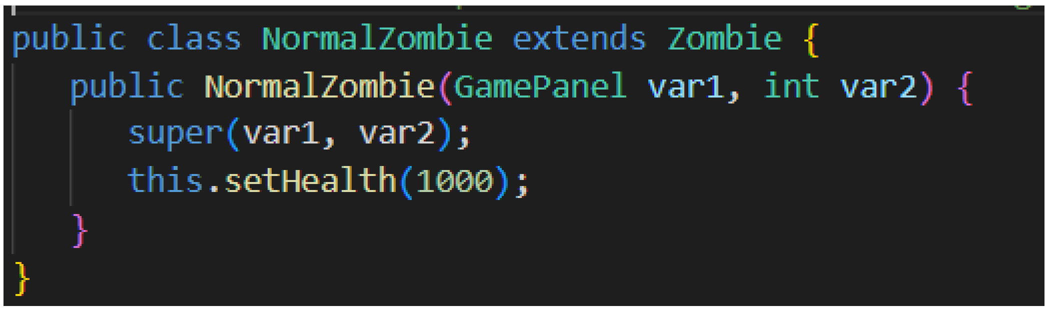
```java
public int getPosX() {
    return this.posX;
}

public void setPosX(int var1) {
    this.posX = var1;
}

public int getMyLane() {
    return this.myLane;
}

public void setMyLane(int var1) {
    this.myLane = var1;
}

public boolean isMoving() {
    return this.isMoving;
}

public void setMoving(boolean var1) {
    this.isMoving = var1;
}

public int getSlowInt() {
    return this.slowInt;
}

public void setSlowInt(int var1) {
    this.slowInt = var1;
}
```



```java
public int getHealth() {
    return this.health;
}

public void setHealth(int var1) {
    this.health = var1;
}

public int getSpeed() {
    return this.speed;
}

public void setSpeed(int var1) {
    this.speed = var1;
}

public GamePanel getGp() {
    return this.gp;
}

public void setGp(GamePanel var1) {
    this.gp = var1;
}
```

b, NormalZombie

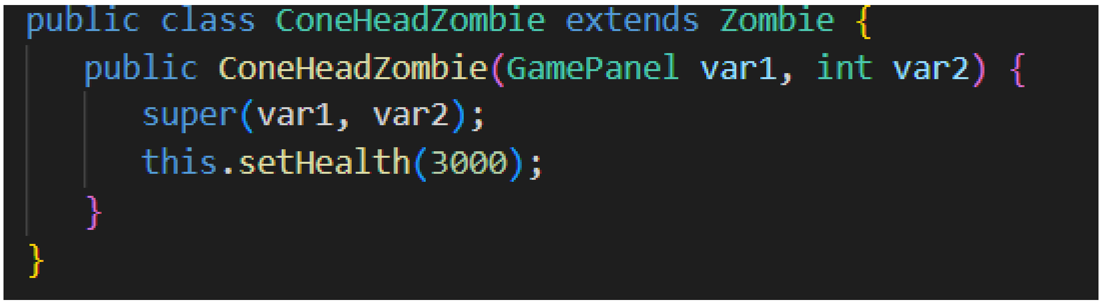
The first subclass of Zombie is NormalZombie, which contains the unique name, health while inheriting the properties from Zombie class.
We set this class as **extends** class of Zombie:



```java
public class NormalZombie extends Zombie {
    public NormalZombie(GamePanel var1, int var2) {
        super(var1, var2);
        this.setHealth(1000);
    }
}
```

- The keyword **extends** allows NormalZombie class accesses all the methods and fields of Zombie (like health, speed, advance(), etc.) unless overridden.
- In the constructor, we use the keyword **super** to call the constructor of the parent class (**Zombie**) and pass var1 (the GamePanel instance) and var2 (the lane number) to it.
- **this.setHealth(1000);:** Explicitly sets the health of the normal zombie to 1000.

c, ConeHeadZombie



```java
public class ConeHeadZombie extends Zombie {
    public ConeHeadZombie(GamePanel var1, int var2) {
        super(var1, var2);
        this.setHealth(3000);
    }
}
```

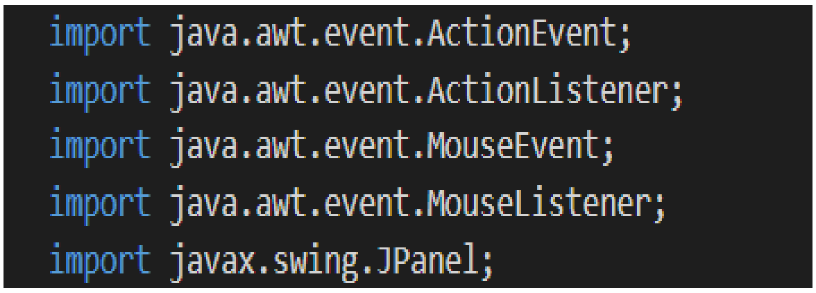
The second subclass of Zombie is ConHeadZombie, which contains the unique name, health while inheriting the properties from Zombie class.
We set this class as **extends** class of Zombie:

- The keyword **extends** allows ConeHeadZombie class accesses all the methods and fields of Zombie (like health, speed, advance(), etc.) unless overridden.
- In the constructor, we use the keyword **super** to call the constructor of the parent class (**Zombie**) and pass var1 (the GamePanel instance) and var2 (the lane number) to it.
- **this.setHealth(1000);:** Explicitly sets the health of the cone head zombie to 1000.

8. Collider

The Collider class represents a **tile or cell** in the game grid where plants can be placed and zombies interact. It extends JPanel (a component from the Swing library) and implements MouseListener to respond to mouse events.

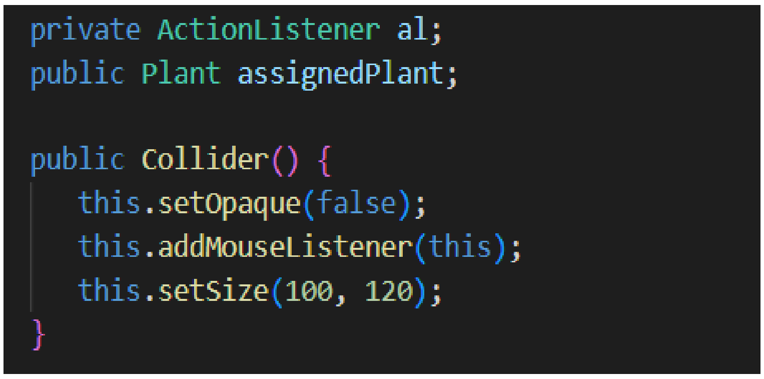
We decided to import components which are available in Java library including



```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.JPanel;
```

ActionEvent/ActionListener, MouseEvent/MouseListener, Jpanel

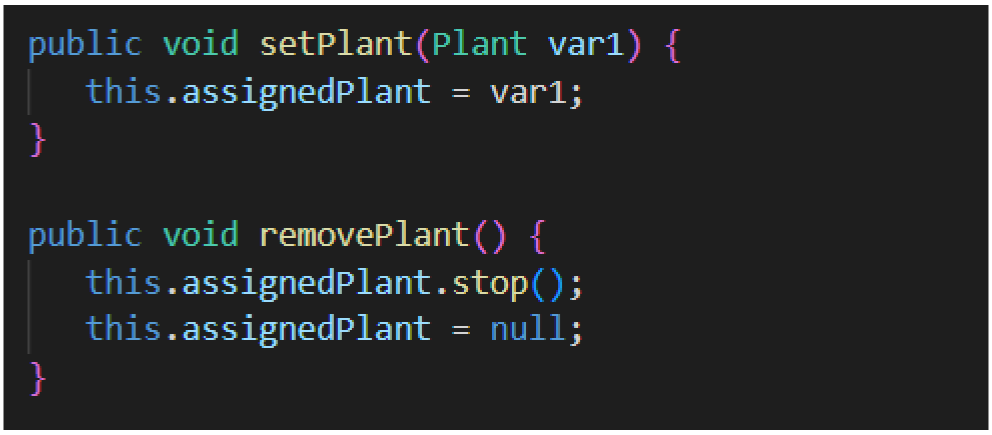
- **al (ActionListener):**
  - An optional action listener that is triggered when certain mouse events (e.g., mouseReleased) occur on this collider.
  - Used to handle game logic, such as selecting or activating a plant.



```java
private ActionListener al;
public Plant assignedPlant;

public Collider() {
    this.setOpaque(false);
    this.addMouseListener(this);
    this.setSize(100, 120);
}
```

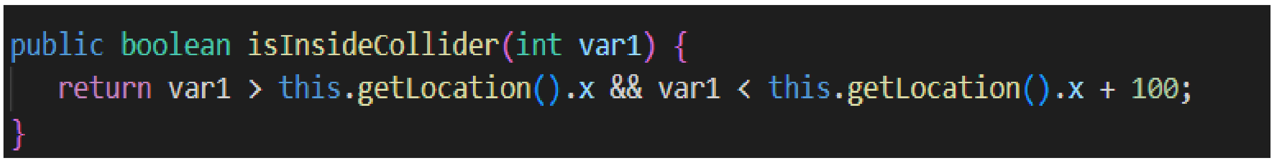
- **assignedPlant (Plant):**
  - Represents the plant currently placed in this collider.
  - If no plant is placed, this is null.
  - Allows interaction between zombies and the plants on the game grid.

- The constructor **Collider()** is use to set the properties of components such as not blocking background, set the MouseListener, and collider size:
  - **this.setOpaque(false):** Makes the panel transparent, so it won't block background visuals.
  - **this.addMouseListener(this):** allows the colliders to reach the mouse interaction
  - **this. setSize(100, 120)**: Sets the size of the collider to a width of 100 pixels and height of 120 pixels



```java
public void setPlant(Plant var1) {
   this.assignedPlant = var1;
}

public void removePlant() {
   this.assignedPlant.stop();
   this.assignedPlant = null;
}
```

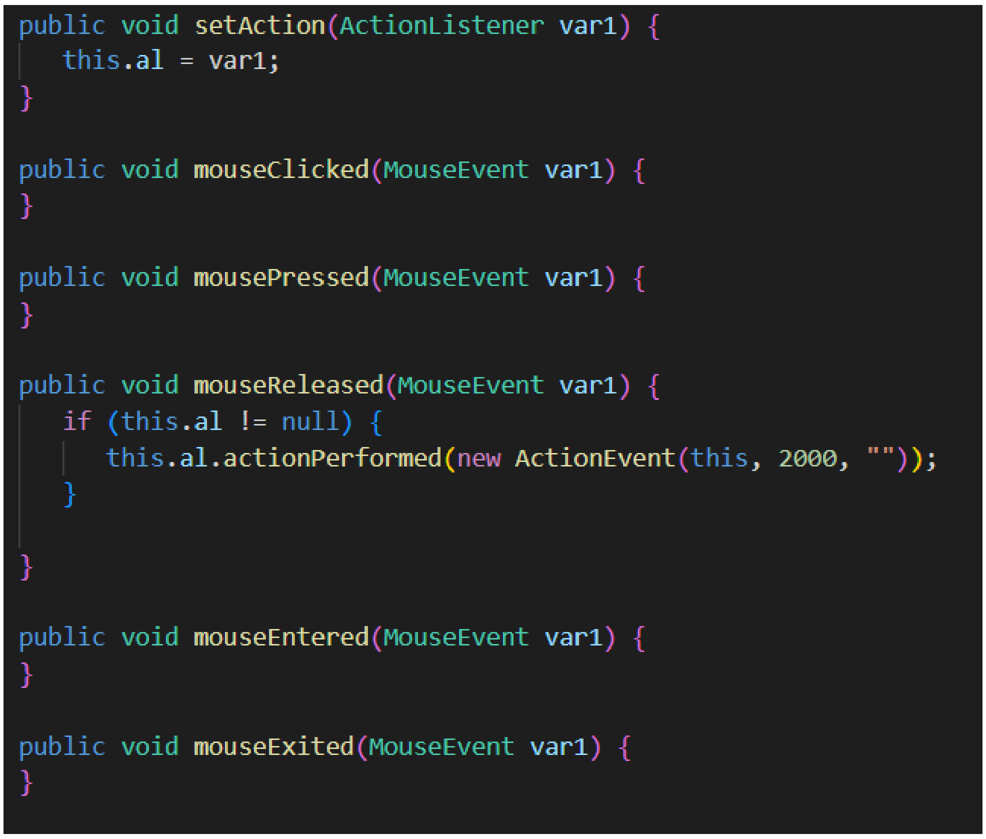
- **`setPlant(Plant var1)`**:
- Assigns a plant to this collider.
- The `Plant` object represents the in-game plant placed on the tile.
- **`removePlant()`**:
- Removes the assigned plant.
- Calls the plant's `stop()` method (likely halting its animations or actions).
- Sets `assignedPlant` to `null`, indicating no plant is present.



```java
public boolean isInsideCollider(int var1) {
   return var1 > this.getLocation().x && var1 < this.getLocation().x + 100;
}
```

The boolean method **isInsideCollider()** is to check if a given position **(var1)** lies within the collider's horizontal bounds. It returns true if the position falls between the left **(getLocation().x)** and right **(getLocation().x + 100)** edges of the collider.

Finally, the set of mouse interactions are use to present the action on the colliders like click, release the mouse button:

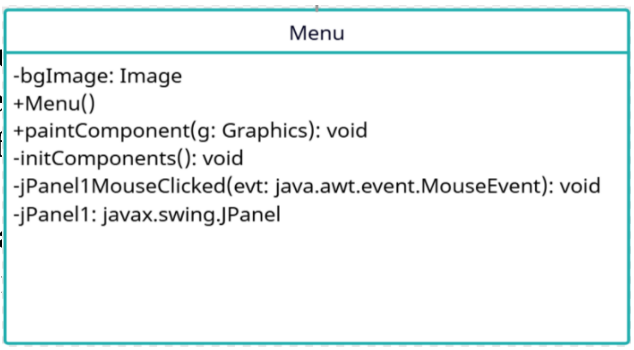


```
public void setAction(ActionListener var1) {
    this.al = var1;
}

public void mouseClicked(MouseEvent var1) {
}

public void mousePressed(MouseEvent var1) {
}

public void mouseReleased(MouseEvent var1) {
    if (this.al != null) {
        this.al.actionPerformed(new ActionEvent(this, 2000, ""));
    }

}

public void mouseEntered(MouseEvent var1) {
}

public void mouseExited(MouseEvent var1) {
}
```

- **setAction(ActionListener var1)**:
- Assigns an ActionListener to this collider.
- The action listener will be triggered when certain mouse events occur.
- **mouseReleased(MouseEvent var1)**:
- If an ActionListener is set, this method triggers its actionPerformed method, creating a new ActionEvent with the collider as the source.
- This is often used to handle user interactions, like planting or selecting a plant.

9. Menu

Menu Class is created to display the menu interface in the "Plant and Zombie" game This class is combined with the display of graphics (wallpapers) and playing background music. Besides that, it also h the function of switching from menu inte to game.



Menu
-bgImage: Image +Menu() +paintComponent(g: Graphics): void -initComponents(): void -jPanel1MouseClicked(evt: java.awt.event.MouseEvent): void -jPanel1: javax.swing.JPanel

To visualize the above functions, our group uses paintComponent to download and display the image, initComponents to manage and create interface components, jPanel1MouseClicked to handle click-and-navigate events. In addition, awt/swing import the Jpanel, Graphic to support creating menu interfaces with backgrounds.

We import:

```
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.event.MouseEvent;
import java.awt.image.ImageObserver;
import javax.swing.GroupLayout;
import javax.swing.ImageIcon;
import javax.swing.JPanel;
import javax.swing.GroupLayout.Alignment;
```

Declaring Image object to store the image used as the background of the menu and SoundPlayer object to display the background music of the menu:

```
public class Menu extends JPanel {
  private Image bgImage;
  private SoundPlayer soundPlayer;
  private JPanel jPanel1;

  public Menu() {
    this.initComponents();
    this.setSize(1012, 785);
    this.bgImage = (new ImageIcon(this.getClass().getResource("images/menu.jpg"))).getImage();
    this.soundPlayer = new SoundPlayer();
    this.soundPlayer.loadSound("sound/background-music.wav");
    this.soundPlayer.loop();
  }
```

- initComponent: initializing interface components
- setSize: set the size of the panel
- bgImage = new ImageIcon(...).getImage(): download the image from menu.jpg file in images folder
- Initialize soundPlayer: download the music file (background-music.wav) and play the background music in the loop mode

Drawing the graphic elements on the Menu panel:

```
protected void paintComponent(Graphics var1) {
    super.paintComponent(var1);
    var1.drawImage(this.bgImage, 0, 0, (ImageObserver)null);
  }
```

- paintComponent: is the function used to draw the menu

- super.paintComponent(g): ensuring the panel draw the basic components correctly
- g.drawImage: drawing the background image at the coordinates (0,0)

Implement the user interaction with the UI:

```
private void initComponents() {
    this.jPanel1 = new JPanel();
    this.setPreferredSize(new Dimension(1012, 785));
    this.jPanel1.setOpaque(false);
    this.jPanel1.addMouseListener(new Menu$1(this));
    GroupLayout var1 = new GroupLayout(this.jPanel1);
    this.jPanel1.setLayout(var1);
    var1.setHorizontalGroup(var1.createParallelGroup(Alignment.LEADING).addGap(0, 387, 32767));
    var1.setVerticalGroup(var1.createParallelGroup(Alignment.LEADING).addGap(0, 116, 32767));
    GroupLayout var2 = new GroupLayout(this);
    this.setLayout(var2);

var2.setHorizontalGroup(var2.createParallelGroup(Alignment.LEADING).addGroup(Alignment.TRAILING, var2.createSequentialGroup().addContainerGap(523, 32767).addComponent(this.jPanel1, -2, -1, -2).addGap(102, 102, 102)));

var2.setVerticalGroup(var2.createParallelGroup(Alignment.LEADING).addGroup(var2.createSequentialGroup().addGap(122, 122, 122).addComponent(this.jPanel1, -2, -1, -2).addContainerGap(547, 32767)));
  }

  private void jPanel1MouseClicked(MouseEvent var1) {
    this.soundPlayer.stop();
    GameWindow.begin();
  }
}
```

- initComponents(): Setting up the layout and look for game

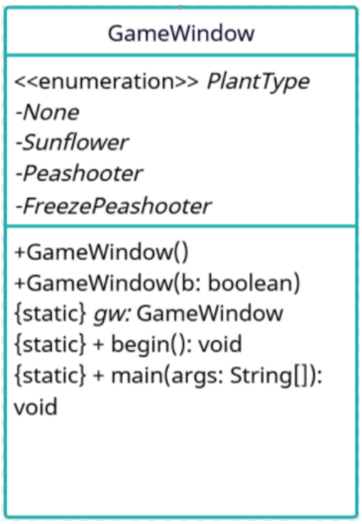
- Using this.jPanel1 = new JPanel() to create jPanel1 as the interface components; this.setPreferredSize(new Dimension(1012, 785)) to set the default size for the entire interface (1012, 785)
- Adding MouseListener into jPanel1 to handle clicks
- Using GroupLayout to layout

+ Elements inside jPanel1 (HorizontalGroup: 387, VerticalGroup: 116)
+ Full interface: jPanel1 is positioned 523px away from the left edge and 102px away from the right edge; and jPanel1 is placed 122px from the top edge and 547px from the bottom edge.

- jPanel1MouseClicked(MouseEvent var1): helping users handle features when entering the game

- this.soundPlayer.stop(): stopping playing music
- GameWindow.begin(): switching from Menu Interface to Play Interface

10. GameWindow



The objective of GameWindow class is to display the main interface for game and gameplay, background music for menu and game, interaction with graphic objects (Plant cards).

To complete those objectives, we apply <<enumeration>>PlantType to define plants in game, and GameWindow() initializes the main interface of the game.

Additionally, javax.sound.sampled serve music purpose (AudioInputStream/AudioSystem/Clip /LineUnavailableException/UnsupportedAudioFileException); javax.swing imports the JFrame, JLabel, JLayeredPane to provide UI component; java.awt.event manages user interaction events (like selecting a plant when tapping on the plant card); java.io manages files (such as downloading the audio file from folder), managing errors when audio file can not be accessed.

We import:

```
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
import javax.sound.sampled.LineUnavailableException;
import javax.sound.sampled.UnsupportedAudioFileException;
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.io.File;
import java.io.IOException;
```

Determining the type of plants:

```
public class GameWindow extends JFrame {


  enum PlantType {
    None,
    Sunflower,
    Peashooter,
    FreezePeashooter
  }
```

- Represent the crops in the game
  - None: No plant is selected
  - SunFlower: Sun flower
  - Peashooter: The plant shoot bean
  - FreezePeashooter: Frozen bean shoots

Managing music:

```
static SoundPlayer soundPlayerMenu;
   static SoundPlayer soundPlayerGame;
// Static sound player for game music

   static GameWindow gw;
```

- soundPlayerMenu: managing background music of menu
- soundPlayerGame: managing background music when playing game

Main interface configuration:

```
public GameWindow() {
       setSize(1012, 785);
       setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
       setLayout(null);
```

- setSize(): setting the size of the game window (width: 1012px, height: 785px)
- setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE): closing the window, the program exits
- setLayout(null): using a free layout , which allows to manually set the position and size of the components

Gaming area

```
JLabel sun = new JLabel("SUN");
       sun.setLocation(37, 80);
       sun.setSize(60, 20);

       GamePanel gp = new GamePanel(sun);
       gp.setLocation(0, 0);
       getLayeredPane().add(gp, new Integer(0));

       setResizable(false);
       setVisible(true);
```

- Displaying “SUN”
  - setLocation(37, 80): setting location and setSize(60, 20): setting size

● Displaying main play area
- Managing the display order of interface components (allow figures to be stacks on top of each other)
● Configure the properties of the game window
- setResizable(false): do not allow user to resize the window
- setVisible(true): showing window

Control music when switching from menu to game

```
// Stop menu music if playing
    if (soundPlayerMenu != null) {
      soundPlayerMenu.stop();
    }

    // Initialize and play game music
    soundPlayerGame = new SoundPlayer();
    soundPlayerGame.loadSound("sound/main-sound.wav");
    soundPlayerGame.loop(); // Loop the music

```

● soundPlayerMenu.stop(): stopping music if it's playing and soundPlayerGame: playing and replaying music

Adding Plant cards

```
PlantCard sunflower = new PlantCard(new
ImageIcon(this.getClass().getResource("images/cards/card_sunflower.png")).getImage());
    sunflower.setLocation(110, 8);
    sunflower.setAction((ActionEvent e) -> {
      gp.setActivePlantingBrush(PlantType.Sunflower);
    });
    getLayeredPane().add(sunflower, new Integer(3));

    PlantCard peashooter = new PlantCard(new
ImageIcon(this.getClass().getResource("images/cards/card_peashooter.png")).getImage());
    peashooter.setLocation(175, 8);
    peashooter.setAction((ActionEvent e) -> {
      gp.setActivePlantingBrush(PlantType.Peashooter);
    });
    getLayeredPane().add(peashooter, new Integer(3));

    PlantCard freezepeashooter = new PlantCard(new
ImageIcon(this.getClass().getResource("images/cards/card_freezepeashooter.png")).getImage());
    freezepeashooter.setLocation(240, 8);
    freezepeashooter.setAction((ActionEvent e) -> {
      gp.setActivePlantingBrush(PlantType.FreezePeashooter);
    });
    getLayeredPane().add(freezepeashooter, new Integer(3));

    getLayeredPane().add(sun, new Integer(2));
    setResizable(false);
    setVisible(true);
  }
```

- PlantCard: representing the plant card in the interface
- new ImageIcon(this.getClass().getResource("...")).getImage(): downloading the picture of the card
- setAction: setting action for the card, when clicked, it will select the corresponding plant.

Switching between the menu interface and the game interface

Initializing a game interface

```
public GameWindow(boolean isMenu) {
    if (isMenu) {
      Menu menu = new Menu();
      menu.setLocation(0, 0);
      setSize(1012, 785);
      setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
      getLayeredPane().add(menu, new Integer(0));
      menu.repaint();
      setResizable(false);
      setVisible(true);
    }
  }
```

- Adding Menu, designing the size of window (width: 1012, height: 785) and closing the window

Starting the game

```
public static void begin() {
    gw.dispose();
    gw = new GameWindow();
  }
```

- gw.dispose(): closing the current interface
- new GameWindow(): creating the game window

Launching the game, starting from the menu

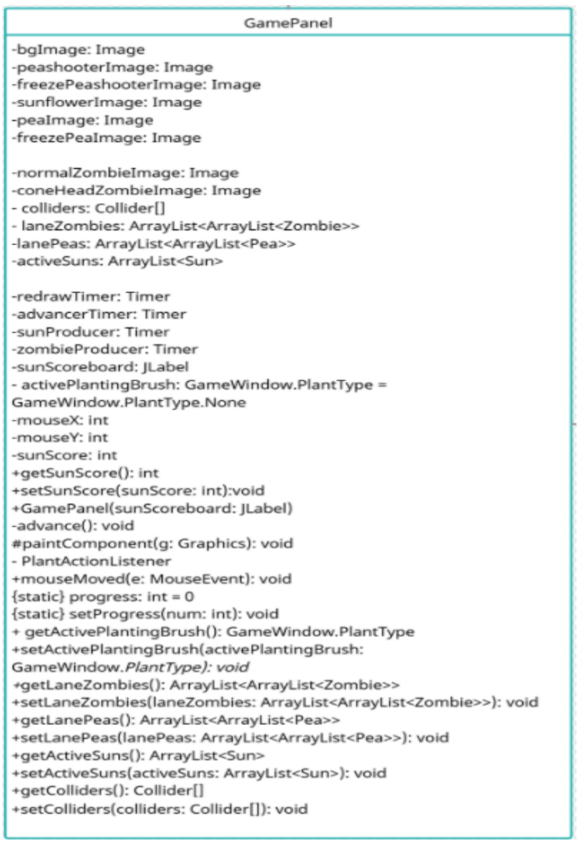
```
public static void main(String[] args) {
        // Initialize menu music
        soundPlayerMenu = new SoundPlayer();
        soundPlayerMenu.loadSound("sound/background-music.wav");
        soundPlayerMenu.loop(); // Loop the music

        // Launch the menu
        gw = new GameWindow(true);
    }
}
```

- Creating and playing music, displaying the interface.

12. GamePanel

GamePanel class serves as a central interface that
manages and processes the logic in the game. This
class integrates the user interface (UI), game objects
(zombies, plants, sun,..) and handles player
interaction.

We apply ArrayList<> to create 2-dimensional list
to manage objects in the game; Timer to manage the
up time of objects; Collider[] represents tiles on the
game board; advance() to proceed with the logic of
the game.



```
GamePanel
-bgImage: Image
-peashooterImage: Image
-freezePeashooterImage: Image
-sunflowerImage: Image
-peaImage: Image
-freezePeaImage: Image

-normalZombieImage: Image
-coneHeadZombieImage: Image
- colliders: Collider[]
- laneZombies: ArrayList<ArrayList<Zombie>>
-lanePeas: ArrayList<ArrayList<Pea>>
-activeSuns: ArrayList<Sun>

-redrawTimer: Timer
-advancerTimer: Timer
-sunProducer: Timer
-zombieProducer: Timer
-sunScoreboard: JLabel
- activePlantingBrush: GameWindow.PlantType =
GameWindow.PlantType.None
-mouseX: int
-mouseY: int
-sunScore: int
+getSunScore(): int
+setSunScore(sunScore: int):void
+GamePanel(sunScoreboard: JLabel)
-advance(): void
#paintComponent(g: Graphics): void
- PlantActionListener
+mouseMoved(e: MouseEvent): void
{static} progress: int = 0
{static} setProgress(num: int): void
+ getActivePlantingBrush(): GameWindow.PlantType
+setActivePlantingBrush(activePlantingBrush:
GameWindow.PlantType): void
+getLaneZombies(): ArrayList<ArrayList<Zombie>>
+setLaneZombies(laneZombies: ArrayList<ArrayList<Zombie>>): void
+getLanePeas(): ArrayList<ArrayList<Pea>>
+setLanePeas(lanePeas: ArrayList<ArrayList<Pea>>): void
+getActiveSuns(): ArrayList<Sun>
+setActiveSuns(activeSuns: ArrayList<Sun>): void
+getColliders(): Collider[]
+setColliders(colliders: Collider[]): void
```

In addition, javax.sound.sampled is used to manage the audio, javax.swing.*/awt.* serve graphics-related tasks, java.awt.event.MouseMotionListener/MouseEvent manages events related to mouse movement, java.io.File manages files (such as downloading the audio file from folder), java.util.ArrayList/Iterator manage lists, java.util.Random handles random values.

We import:

```
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
import java.io.File;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Random;
```

Storing the game's visual resources

```
public class GamePanel extends JLayeredPane implements MouseMotionListener {

   private Image bgImage;
   private Image peashooterImage;
   private Image freezePeashooterImage;
   private Image sunflowerImage;
   private Image peaImage;
   private Image freezePeaImage;

   private Image normalZombieImage;
   private Image coneHeadZombieImage;
   private Image BucketheadZombieImage;


   private Image normalZombieEatImage;
   private Image coneHeadZombieEatImage;
   private Image BucketheadZombieEatImage;
```

- GamePanel extends JLayredPane to support multi-layer management of graphics
- MouseMotionListener: following and serving mouse events

Interactive tool and game logic

```
  private Collider[] colliders;

  private ArrayList<ArrayList<Zombie>> laneZombies;
  private ArrayList<ArrayList<Pea>> lanePeas;
  private ArrayList<Sun> activeSuns;
```

- colliers: using to attach the plant when player chooses
- laneZombies/Peas: storing lists of zombies and bullets
- activeSuns: storing the falling-suns

Timer

```
  private Timer redrawTimer;
  private Timer advancerTimer;
  private Timer sunProducer;
  private Timer zombieProducer;
```

- redrawTimer: updating the screen; advancerTimer: adjusting actions of zombies, peas, suns; sun/zombieProducer: producing random new sun/zombie.

Managing sun score

```
private JLabel sunScoreboard;
  private GameWindow$PlantType activePlantingBrush;
  private int mouseX;
  private int mouseY;
  private int sunScore;
  static int progress = 0;

  public int getSunScore() {
    return this.sunScore;
  }

  public void setSunScore(int var1) {
    this.sunScore = var1;
    this.sunScoreboard.setText(String.valueOf(var1));
  }
```

- JLabel sunScoreboard:

- getSunScore(): showing sun score on the game interface and checking if the player has enough score to buy the plant

- setSunScore(int var1): when sun score changes, this label is also updated the new score
- GameWindow$PlantType activePlantingBrush: representing the type of plants that the player is choosing to plant, its function determines the player's action when clicking on the plant area
- mouseX/Y: storing mouse's coordinates, which is used to display the image of the selected plant according to the mouse position
- static int progress: storing the progress of the game (ex: amount of zombies that the player kills)

Zombies bite sound

```
private static final String BITE_SOUND_PATH = "sound/bite.wav";
public void playBiteSound() {
    try {
      File var1 = new File("sound/bite.wav");
      if (!var1.exists()) {
        System.err.println("Bite sound file not found: sound/bite.wav");
        return;
      }

      AudioInputStream var2 = AudioSystem.getAudioInputStream(var1);
      Clip var3 = AudioSystem.getClip();
      var3.open(var2);
      var3.start();
    } catch (Exception var4) {
      System.err.println("Error playing bite sound: " + var4.getMessage());
    }
  }
```

- BITE_SOUND_PATH: helping increase readability and standardize file management to save the path of the audio file
- playBiteSound(): checking the existence of audio file and play it, if it has the error during playback, the method will process and display an error message
  - File var1 = new File("sound/bite.wav"): references to audio file
  - if (!var1.exists()): checking the existence of sound/bite.wav link in audio file
  - AudioInputStream: reading audio from the file
  - var3.start(): playing audio that is downloaded
  - var4.getMessage(): displaying the error

Initiating size and event

```
public GamePanel(JLabel var1) {
    this.activePlantingBrush = PlantType.None;
    this.setSize(1000, 752);
    this.setLayout((LayoutManager)null);
    this.addMouseMotionListener(this);
    this.sunScoreboard = var1;
    this.setSunScore(50);
```

- Display screen size: 1000 x 752
- Initial score: 50 (enough to plant sunflower)

Downloading image

```
this.bgImage = (new ImageIcon(this.getClass().getResource("images/mainBG.png"))).getImage();
    this.peashooterImage = (new ImageIcon(this.getClass().getResource("images/plants/peashooter.gif"))).getImage();
    this.freezePeashooterImage = (new ImageIcon(this.getClass().getResource("images/plants/freezepeashooter.gif"))).getImage();
    this.sunflowerImage = (new ImageIcon(this.getClass().getResource("images/plants/sunflower.gif"))).getImage();
    this.peaImage = (new ImageIcon(this.getClass().getResource("images/pea.png"))).getImage();
    this.freezePeaImage = (new ImageIcon(this.getClass().getResource("images/freezepea.png"))).getImage();
    this.normalZombieImage = (new ImageIcon(this.getClass().getResource("images/zombies/zombievip.gif"))).getImage();
    this.normalZombieEatImage = (new ImageIcon(this.getClass().getResource("images/zombies/ZombieVipEat.gif"))).getImage();
    this.BucketheadZombieImage = (new ImageIcon(this.getClass().getResource("images/zombies/BucketheadZombie.gif"))).getImage();
    this.BucketheadZombieEatImage = (new ImageIcon(this.getClass().getResource("images/zombies/BucketheadZombieAttackp.gif"))).getImage();
    this.coneHeadZombieImage = (new ImageIcon(this.getClass().getResource("images/zombies/ConeheadZombie.gif"))).getImage();
    this.coneHeadZombieEatImage = (new ImageIcon(this.getClass().getResource("images/zombies/ConeheadZombieAttack.gif"))).getImage();
```

- ImageIcon: downloading the image from the folder

Initializing rows

```
    this.laneZombies = new ArrayList();
    this.laneZombies.add(new ArrayList());
    this.laneZombies.add(new ArrayList());
    this.laneZombies.add(new ArrayList());
    this.laneZombies.add(new ArrayList());
    this.laneZombies.add(new ArrayList());
    this.lanePeas = new ArrayList();
    this.lanePeas.add(new ArrayList());
    this.lanePeas.add(new ArrayList());
    this.lanePeas.add(new ArrayList());
    this.lanePeas.add(new ArrayList());
    this.lanePeas.add(new ArrayList());
    this.activeSuns = new ArrayList();
```

- laneZombies/Peas using ArrayList to create list, each list represents a zombie/peas in a specific row (there are 5 rows in total)

Initializing cells

```
this.colliders = new Collider[45];

    for(int var2 = 0; var2 < 45; ++var2) {
      Collider var3 = new Collider();
      var3.setLocation(44 + var2 % 9 * 100, 109 + var2 / 9 * 120);
      var3.setAction(new GamePanel$PlantActionListener(this, var2 % 9, var2 / 9));
      this.colliders[var2] = var3;
      this.add(var3, new Integer(0));
    }
```

- Collier represents cell (5 rows x 9 columns = 45 cells).
- Formula: x = 44 + (column index × 100) & y = 109 + (row index × 120)

Timer

```
this.redrawTimer = new Timer(16, (var1x) -> {
      this.repaint();
    });
    this.redrawTimer.start();
    this.advancerTimer = new Timer(60, (var1x) -> {
      this.advance();
    });
    this.advancerTimer.start();
    this.sunProducer = new Timer(4000, (var1x) -> {
      Random var2 = new Random();
      Sun var3 = new Sun(this, var2.nextInt(800) + 100, 0, var2.nextInt(300) + 200);
      this.activeSuns.add(var3);
      this.add(var3, new Integer(1));
    });
    this.sunProducer.start();
    this.zombieProducer = new Timer(7000, (var1x) -> {
      Random var2 = new Random();
      LevelData var3 = new LevelData();
      String[] var4 = LevelData.LEVEL_CONTENT[Integer.parseInt(LevelData.LEVEL_NUMBER) -
1];
      int[][] var5 = LevelData.LEVEL_VALUE[Integer.parseInt(LevelData.LEVEL_NUMBER) - 1];
      int var6 = var2.nextInt(5);
      int var7 = var2.nextInt(100);
      Zombie var8 = null;

      for(int var9 = 0; var9 < var5.length; ++var9) {
        if (var7 >= var5[var9][0] && var7 <= var5[var9][1]) {
          var8 = Zombie.getZombie(var4[var9], this, var6);
        }
      }

      ((ArrayList)this.laneZombies.get(var6)).add(var8);
    });
    this.zombieProducer.start();
  }
```

- this.redrawTimer = new Timer(16, (var1x) -> { this.repaint(); }): Refresh the screen every 16ms for smooth animations
- this.advancerTimer = new Timer(60, (var1x) -> { this.advance(); }): updating game state (e.g: movement of zombies and peas) every 60ms
- this.zombieProducer = new Timer(7000, (var1x): producing zombies every 7 seconds. The type of zombies and the row are randomly selected

Handling status updates for each object in the game: zombies, suns, peas

```
private void advance() {
    int var1;
    for(var1 = 0; var1 < 5; ++var1) {
      Iterator var2 = ((ArrayList)this.laneZombies.get(var1)).iterator();

      while(var2.hasNext()) {
        Zombie var3 = (Zombie)var2.next();
        var3.advance();
      }

      for(int var4 = 0; var4 < ((ArrayList)this.lanePeas.get(var1)).size(); ++var4) {
        Pea var5 = (Pea)((ArrayList)this.lanePeas.get(var1)).get(var4);
        var5.advance();
      }
    }
    for(var1 = 0; var1 < this.activeSuns.size(); ++var1) {
      ((Sun)this.activeSuns.get(var1)).advance();
    }

  }
```

- Iterator var2 … var3.advance(): moving the zombies closer towards the left side of the screen
- for (int var4 = 0…) … var5.advance(): each pea moves forward in its row
- for(var1 0 =…) … ((sun))…).advance(): updating the sun's falling state

Drawing game elements onto the screen

```
protected void paintComponent(Graphics var1) {
    super.paintComponent(var1);
    var1.drawImage(this.bgImage, 0, 0, (ImageObserver)null);
```

- paintComponent: responsible for drawing the entire interface for the component and used in game to draw elements such as: background, zombies, plants, peas,..

- super.paintComponent(var1): cleaning the drawing are before drawing on the new component

- var1.drawImage(this.bgImage, 0, 0, (ImageObserver)null): displaying background

```
 int var2;
    for(var2 = 0; var2 < 45; ++var2) {
      Collider var3 = this.colliders[var2];
      if (var3.assignedPlant != null) {
        Plant var4 = var3.assignedPlant;
        if (var4 instanceof Peashooter) {
          var1.drawImage(this.peashooterImage, 60 + var2 % 9 * 100, 129 + var2 / 9 * 120,
(ImageObserver)null);
        }

        if (var4 instanceof FreezePeashooter) {
          var1.drawImage(this.freezePeashooterImage, 60 + var2 % 9 * 100, 129 + var2 / 9 * 120,
(ImageObserver)null);
        }

        if (var4 instanceof Sunflower) {
          var1.drawImage(this.sunflowerImage, 60 + var2 % 9 * 100, 129 + var2 / 9 * 120,
(ImageObserver)null);
        }
      }
    }
```

- Drawing types of plants
- this.colliders: an array contains Collider object, each representing a cell in a grid, each collier may or may not contain a type of plant (assignedPlant object), and iterate through all 45 cells in the grid
- if (var4 instanceof ...) {
  var1.drawImage(this...., 60 + var2 % 9 * 100, 129 + var2 / 9 * 120, (ImageObserver)null);}: types of plants

- Column: 60 + var2 % 9 * 100 (located x based on column)
- Row: 129 + var2 / 9 * 120 (located y based on row)

```
for(var2 = 0; var2 < 5; ++var2) {
      Iterator var5 = ((ArrayList)this.laneZombies.get(var2)).iterator();

      while(var5.hasNext()) {
        Zombie var7 = (Zombie)var5.next();
        if (var7 instanceof NormalZombie) {
          if (!var7.isAttacking()) {
            var1.drawImage(this.normalZombieImage, var7.getPosX(), 109 + var2 * 120, (ImageObserver)null);
          } else {
            var1.drawImage(this.normalZombieEatImage, var7.getPosX(), 109 + var2 * 120, (ImageObserver)null);
          }
        } else if (var7 instanceof ConeHeadZombie) {
          if (!var7.isAttacking()) {
            var1.drawImage(this.coneHeadZombieImage, var7.getPosX(), 109 + var2 * 120, (ImageObserver)null);
          } else {
            var1.drawImage(this.coneHeadZombieEatImage, var7.getPosX(), 109 + var2 * 120, (ImageObserver)null);
          }
        } else if (var7 instanceof BucketheadZombie) {
          if (!var7.isAttacking()) {
            var1.drawImage(this.BucketheadZombieImage, var7.getPosX(), 109 + var2 * 120, (ImageObserver)null);
          } else {
            var1.drawImage(this.BucketheadZombieEatImage, var7.getPosX(), 109 + var2 * 120, (ImageObserver)null);
          }
        }
      }
```

- Drawing types of zombies
- for(var2 = 0; var2 < 5; ++var2) {
  Iterator var5 = ((ArrayList)this.laneZombies.get(var2)).iterator();

- A list (ArrayList) contains 5 elements, each element is a list of zombie in each lane

- Zombie var7 = (Zombie)var5.next(); ... var1.drawImage(this....Image, var7.getPosX(), 109 + var2 * 120, (ImageObserver)null); } }: types of zombies

- !var7.isAttacking(): if zombies do not attack, drawing a moving state image/ if zombies attack, drawing a eating state image

- Coordinates:

- Horizontal position (x): taken from var7.getPosX() (current position of zombies)
- Vertical position (y): 109 + var2 * 120 (determine the lane position based on var2)

```
for(int var6 = 0; var6 < ((ArrayList)this.lanePeas.get(var2)).size(); ++var6) {
        Pea var8 = (Pea)((ArrayList)this.lanePeas.get(var2)).get(var6);
        if (var8 instanceof FreezePea) {
          var1.drawImage(this.freezePeaImage, var8.getPosX(), 130 + var2 * 120, (ImageObserver)null);
        } else {
          var1.drawImage(this.peaImage, var8.getPosX(), 130 + var2 * 120, (ImageObserver)null);
        }
      }
    }
```

- Drawing peas
- (ArrayList)this.lanePeas: a list contains peas in each lane
- Types of peas
- Coordinates:

- Horizontal position (x): taken from var8.getPosX()
- Vertical position (y): 130 + var2 * 120 (determine the lane based on var2)

```
if (!"".equals(this.activePlantingBrush) && this.activePlantingBrush == PlantType.Sunflower) {
      var1.drawImage(this.sunflowerImage, this.mouseX, this.mouseY, (ImageObserver)null);
    }


  }
```

- Displaying plants selected to grow

- activePlantingBrush: type of plants that selected to grow
- this.mouseX/Y: displaying the corresponding image at the mouse position

Managing the progress: changing the level or ending the game

```
public static void setProgress(int var0) {
    progress += var0;
    System.out.println(progress);
    if (progress >= 200) {
      if ("1".equals(LevelData.LEVEL_NUMBER)) {
        JOptionPane.showMessageDialog((Component)null,      "LEVEL_CONTENT      Completed
!!!\nStarting next LEVEL_CONTENT");
        GameWindow.gw.dispose();
        LevelData.write("2");
        GameWindow.gw = new GameWindow();
      } else {
        JOptionPane.showMessageDialog((Component)null,      "LEVEL_CONTENT      Completed
!!!\nMore Levels will come soon !!!\nResetting data");
        LevelData.write("1");
        System.exit(0);
      }

      progress = 0;
    }

  }
```

● progress += var0: increasing the progress (e.g: when killing the zombie)
● progress >= 200
- if ("1".equals(LevelData.LEVEL_NUMBER)): current level equal to 1
+ JOptionPane: displaying level completion notification
+ GameWindow.gw.dispose(): closing the current game window
+ LevelData.write("2"): writing the new level '2' to the data
+ GameWindow.gw = new GameWindow(): initializing the new game window
- If not level 1: displaying level completion notification and announcement about more levels coming in the future
+ LevelData.write("1"): resetting the level date to 1 and System.exit(0): exiting the program

Providing access to important data structures such as laneZombies, lanePeas, activeSuns and colliders (getter and setter)

```
public GameWindow$PlantType getActivePlantingBrush() {
    return this.activePlantingBrush;
  }
  public void setActivePlantingBrush(GameWindow$PlantType var1) {
    this.activePlantingBrush = var1;
  }

  public ArrayList<ArrayList<Zombie>> getLaneZombies() {
    return this.laneZombies;
  }

  public void setLaneZombies(ArrayList<ArrayList<Zombie>> var1) {
    this.laneZombies = var1;
  }

  public ArrayList<ArrayList<Pea>> getLanePeas() {
    return this.lanePeas;
  }

  public void setLanePeas(ArrayList<ArrayList<Pea>> var1) {
    this.lanePeas = var1;
  }

  public ArrayList<Sun> getActiveSuns() {
    return this.activeSuns;
  }

  public void setActiveSuns(ArrayList<Sun> var1) {
    this.activeSuns = var1;
  }

  public Collider[] getColliders() {
    return this.colliders;
  }

  public void setColliders(Collider[] var1) {
    this.colliders = var1;
  }
}
```

CHAP 4:
FINAL APP GAME

Source code (link github):

https://github.com/trannhat900/pvz/

Demo video:

[UPDATE LATER]

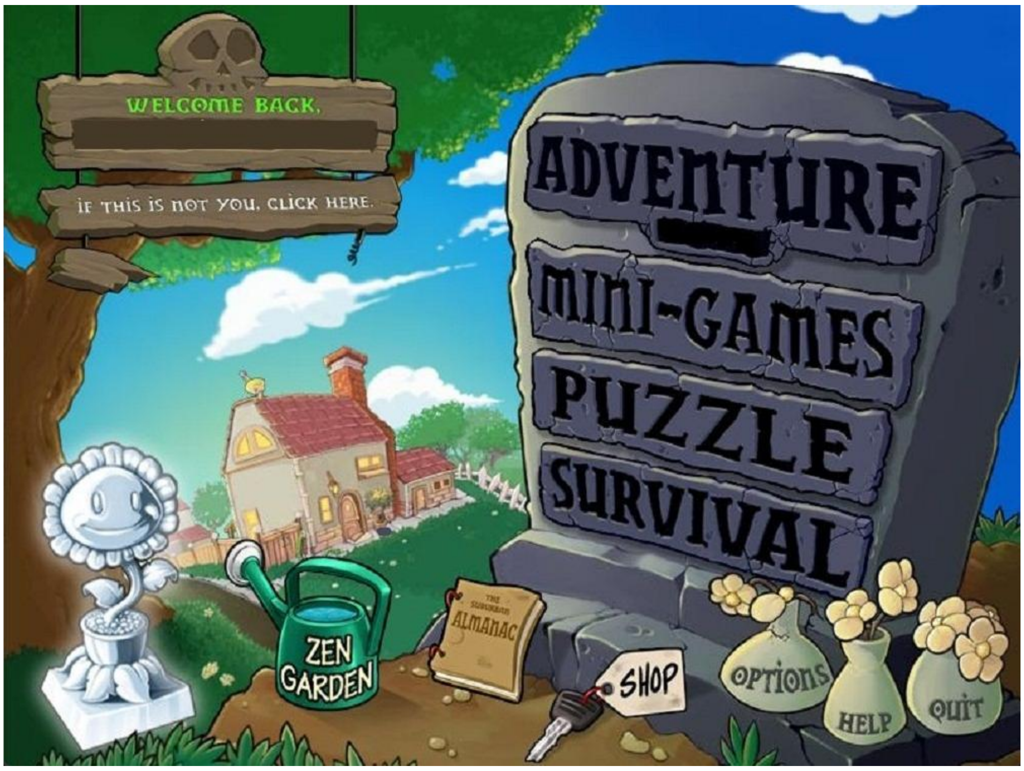
Instruction

1. Begin the game:

Click on the to open the game. Main menu screen will appear as below:

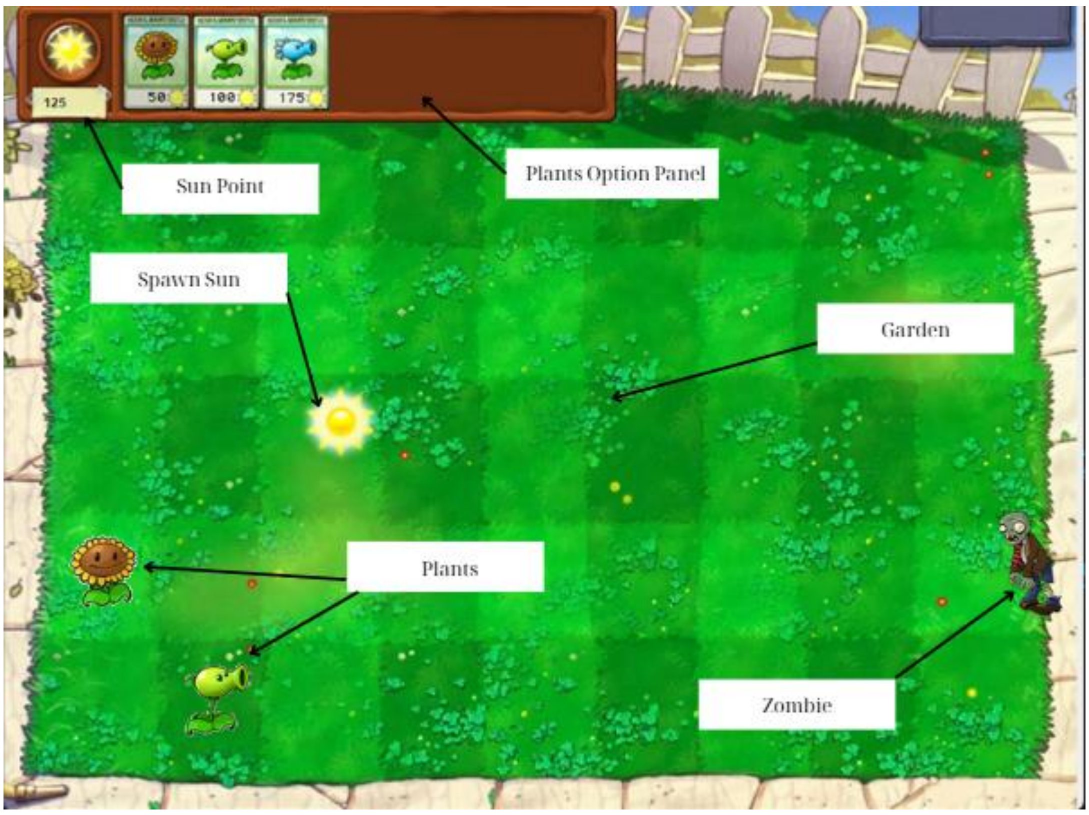
Click on the “Adventure” button to start the game

2. How to play

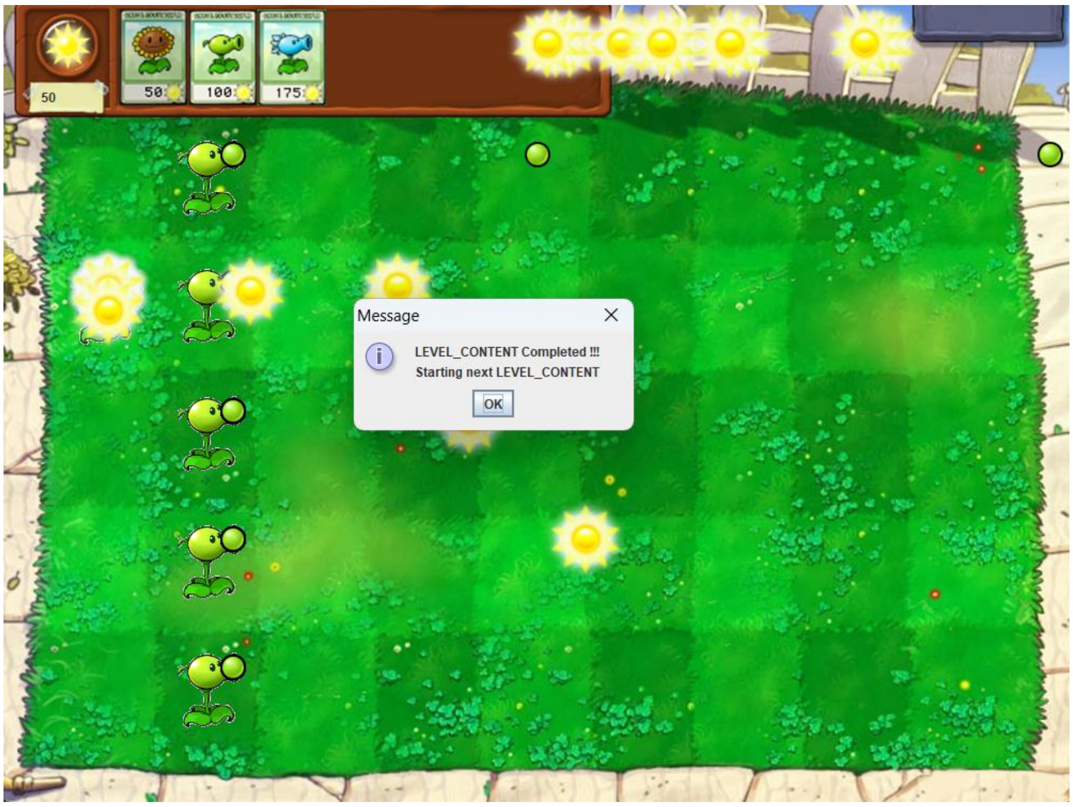


The rule of the game is very simple:

- Each plant has a specific cost and ability, player can drag and drop a plant from the Plants Option Panel to the garden if its cost is less than or equal to the current Sun Point.
- To earn the point, use the Sun Flower or collect the spawn Sun



- Case Win: the last zombie is defeated

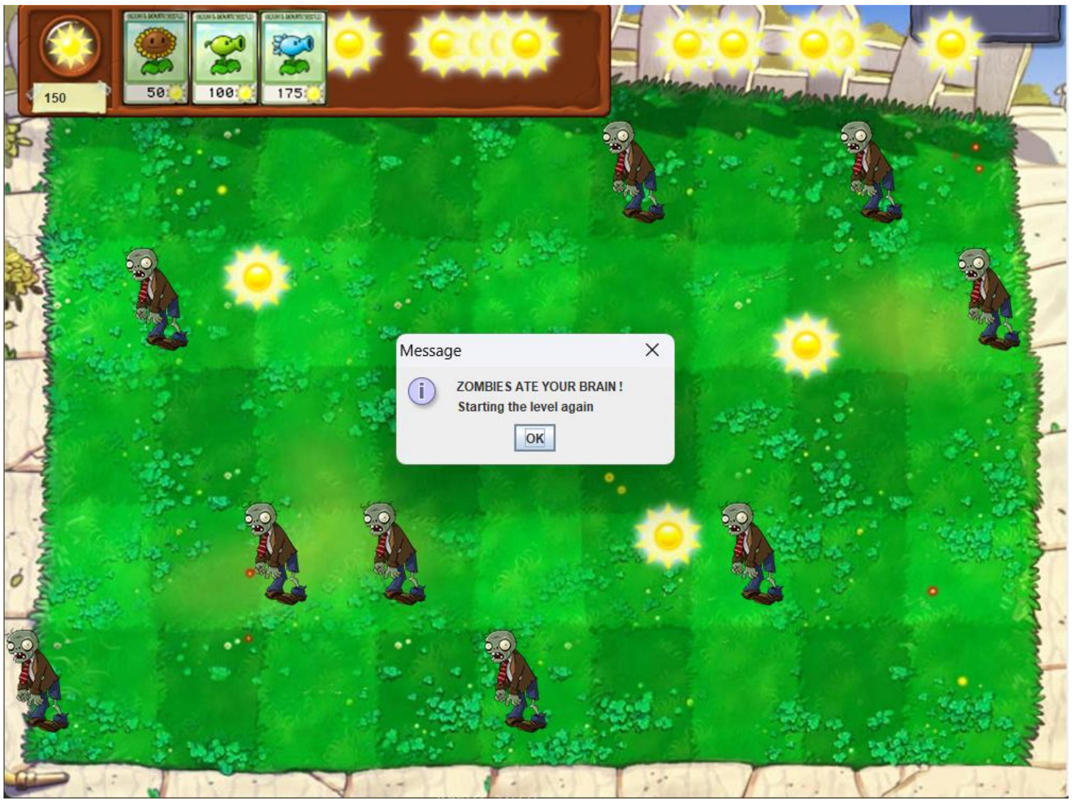


Message

LEVEL_CONTENT Completed !!!
Starting next LEVEL_CONTENT

OK

- Case Lose: a zombie reach player’s house



Message

ZOMBIES ATE YOUR BRAIN !
Starting the level again

OK

CHAP 5: EXPERIENCE

After a month working with this project, we can conclude with the point: Game development not only provides opportunity to practice, but also enhance the linking in a group, leading to better cooperating.
We can finally finish the project with so many difficult things which we had to overcome to gain the success. We had to find the sample source code, study extra contents for OOP on Youtube and practice our programming skills from the easiest exercise to the most difficult program. And we realize that Object-Oriented Programming students should self-study most of the time. Finally, the IT world is too big to wait and just learn from school.