REPORT OOP

Members:

Trần Đình Hưng	ITCSIU24029
Hồ Gia Huy	ITDSIU23007

ONLY APOCALYPTIC SURVIVOR

REPORT OOP

-Game title: Only apocalyptic survivor

OVERVIEW

This is a report about our 2D game project. With many efforts, we only can develop a simple survival game. In this report, we will introduce our game through 2 sections:

Section I: Game rules and features

Section II: System design.

SECTION I: GAME RULES AND FEATURES

- "Only Apocalyptic Survivor" is a survival-themed game where the player's primary objective is to eliminate as many monsters as possible while keeping their health above zero—if the player's health reaches zero, the game ends.

- At the start of the game, a main menu is displayed with two primary options: Start Game and Quit. Selecting Start Game will initiate the gameplay, while choosing Quit will terminate the application by closing the Java Virtual Machine (JVM).

- After entering the game, the player is equipped with a basic sword and shield. The next goal is to interact with the nearby NPC standing next to a chest (by going close to the NPC and press Enter or Z) and carefully follow the instructions provided by the NPC which are: find the hidden key to open a chest which include a better sword and shield, then find and kill a skeleton to start the game.

- The game features an infinite number of rounds, with each round consisting of 10 enemies—5 slimes and 5 skeletons. Slimes are capable of launching long-range attacks against the player, but they have lower overall stats compared to skeletons. After completing each round, the monsters' stats are slightly increased to gradually raise the difficulty level.

- Scattered throughout the map are several holes; if the player's character falls into one, they will lose a portion of their health.

- The game features an inventory system that allows players to manage the items their character possesses. Pressing TAB opens the inventory menu, and items can be used or equipped by pressing Enter.

- Pressing TAB not only opens the inventory but also displays the character's basic statistics, such as health, strength, defense, exp and mana.

- The game features a single skill: Fireball Shooting. The player can activate this ability by pressing the F key, allowing them to attack enemies from a long range.

- There are several items in the game:

1. Strength potion: When the character picks up a Strength Potion, it is automatically added to the inventory. If the player opens the inventory and selects the potion, the character's strength attribute will be increased.
2. Boots: When the character picks up a Boots, his speed will be increased
3. Sword: 2 kinds of sword(basic and ultra one)
4. Shield: 2 kinds of shield (basic and ultra one)
5. Exp: When the character collects an experience item, their EXP increases by 1. Once the accumulated EXP reaches the required threshold specified under "EXP needed to next level," the character's level will increase by 1.
6. Key: use to open chest

SECTION II: SYSTEM DESIGN

2.1 Set up game:

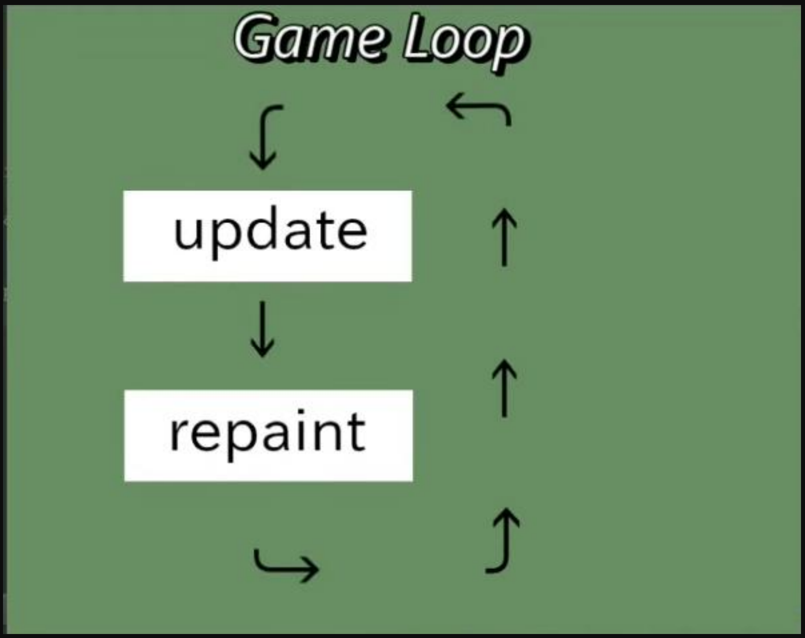
- Before the game officially begins its execution loop, the setup() method in the GamePanel class is invoked to initialize all core components required for gameplay. This preparatory step ensures that all necessary assets and elements are loaded and properly positioned before the first frame is rendered.

- The setup() method performs the following key tasks:

- Background Music Initialization: It configures and starts the background music or ambient sound to ensure the game has audio from the beginning.
- Spawning of Game Entities: This includes placing NPCs and monsters at predefined locations on the map, ensuring they are ready to interact with the player.
- Setting Initial States: Game states is set to title state.

2.2 Game loop:

- At the heart of the game's functionality lies a continuous game loop, implemented within the run() method of the GamePanel class. This loop is responsible for maintaining a consistent frame rate and ensuring that the game world is updated and rendered smoothly in real-time.



- The draw() and update() methods in the GamePanel class work closely together to handle the game's visual output and animation logic.

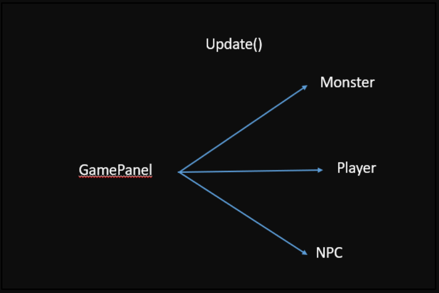
➔ Within the main game loop, the update() and draw() methods of the GamePanel class are invoked on every iteration.

Note: To call method draw() in run(), we have to call repaint() instead of draw()

- To ensure smooth gameplay, the loop is configured to execute 60 times per second, meaning both update() and draw() are each called 60 times every second. This effectively sets the game's frame rate at 60 FPS (frames per second)

- In the draw() method, various images (such as sprites, backgrounds, and UI elements) are assigned to BufferedImage variables. These images represent the current visual state of game entities like the player, monsters, and tiles. By separating these into image buffers, the game can efficiently render the necessary graphics each frame.

- The update() method plays a crucial role in controlling the animation flow. It utilizes internal counters—typically integer variables incremented every frame—to determine which frame of an animation should be shown.



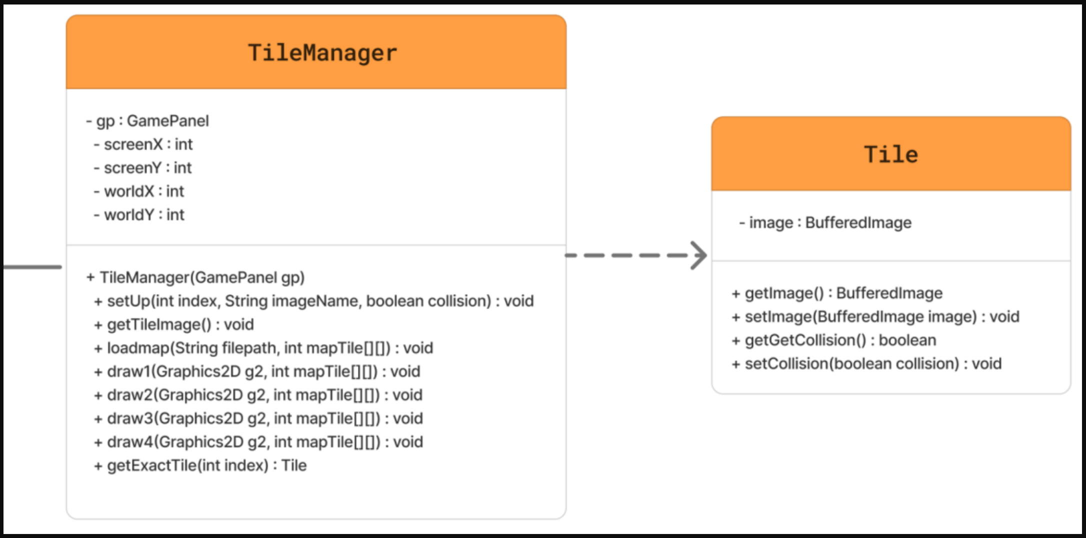
- In this game, each entity—such as the player, NPCs, monsters, and player—exhibits unique behaviors and interactions. To handle these differences effectively, each category of entity is implemented with its own update() method that defines its behavior during each frame.

2.3 Game State:

-To organize the flow and user interactions effectively, the game is structured into multiple distinct states, each representing a different mode or context of gameplay. These states control what the game displays, how it behaves, and which actions are available to the player at any given moment.

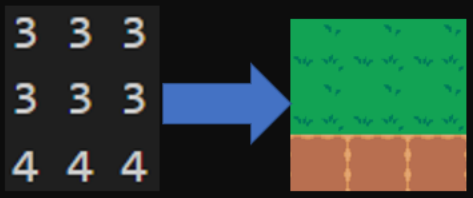
- Title state: This is the initial screen shown when the game is launched. It displays the main menu with options such as "Start Game" and "Quit." No gameplay actions are active during this state.
- Play state: The main gameplay mode. All entities are updated and drawn, user input is processed, and the player can explore the world, fight monsters, interact with NPCs, and progress through the game.
- Pause state: When the player pauses the game (by pressing the key P) , entity updates are halted but the screen is still rendered. This state allows the player to take a break without losing progress.
- Dialogue state: Activated when the player interacts with NPCs, encounters events. During this state, a text box appears on the screen to display scripted dialogues, and player movement is temporarily disabled.
- Character state: This state is triggered when the player opens the inventory or character stats screen (by pressing TAB). It displays attributes, items, and equipment, and allows item usage or equipment changes.
- Overstate: Represents the Game Over screen, which appears when the player's health drops to zero. The game halts and displays a message, typically with options to restart, respawn(only 3 chances) or quit the game.

2.4 Tile management:


TileManager
- gp : GamePanel
- screenX : int
- screenY : int
- worldX : int
- worldY : int
+ TileManager(GamePanel gp)
+ setUp(int index, String imageName, boolean collision) : void
+ getTileImage() : void
+ loadmap(String filepath, int mapTile[][]) : void
+ draw1(Graphics2D g2, int mapTile[][]) : void
+ draw2(Graphics2D g2, int mapTile[][]) : void
+ draw3(Graphics2D g2, int mapTile[][]) : void
+ draw4(Graphics2D g2, int mapTile[][]) : void
+ getExactTile(int index) : Tile
Tile
- image : BufferedImage
+ getImage() : BufferedImage
+ setImage(BufferedImage image) : void
+ getGetCollision() : boolean
+ setCollision(boolean collision) : void

To manage graphical assets efficiently, especially for tile rendering, the game uses an array of BufferedImage objects with fixed size (48x48). Each image in this array represents a unique tile or visual element used to construct the game world.

- Tile management is a fundamental component of this game's map and world design. The game utilizes a tile-based system, where the game world is divided into small, square units called tiles. Each tile typically represents a 2D block of the environment, such as grass, water, trees, or wall.

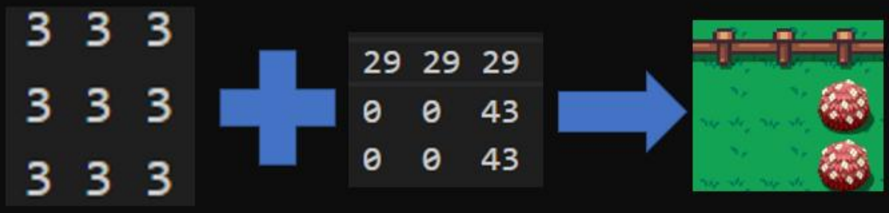


```
3 3 3
3 3 3
4 4 4
```

- Map loading: Map layouts are typically loaded from external files (txt format) where each number represents a tile index. The game reads this file into a 2D array to generate the world map during the setup phase.

- In addition to base tiles that form the ground or static background of the game world, the game also implements a system of overlay tiles. Overlay tiles are an essential graphical enhancement that adds visual depth, interactivity, and realism to the tile-based map.

- Overlay tiles are additional graphical layers drawn on top of the base tile layer, typically used to represent objects that exist above or partially above the ground level, such as: trees, fences,…



```
3 3 3
3 3 3
3 3 3
```

```
29 29 29
0  0  43
0  0  43
```

3: grass

29: fence

43: red leafs

0: null

- Tiles are assigned collision attributes. For example:

- Grass or dirt tiles may allow movement.
- Fences or water tiles may be marked as solid and block movement.

- Efficient Rendering (draw1() – draw2() – draw3() – draw4()):

- To optimize performance, only the tiles within the visible screen (or slightly beyond) are drawn each frame. This prevents unnecessary rendering of off-screen tiles and keeps the game responsive.
- In addition to the basic tile management system, this game implements a direction-aware tile rendering mechanism to ensure smooth visual performance and prevent graphical glitches—especially when the player is in motion. For instance, when player is moving to the right (Pressing D), the tiles will be rendered from left to right,

2.5 Transition from world coordinate to screen coordinate:

a) Definition:

- World Coordinates represent the absolute position of an entity or tile within the overall game map. For example, if the map is 100 tiles wide, a tree might exist at world coordinate (70 * tileSize, 40 * tileSize).

- Screen coordinates refer to the position on the player's visible screen.

b) Why Conversion Is Necessary:

- The player character has to be drawn at the center of the screen, and the rest of the game world scrolls around them. Thus, all tiles and objects must be translated from their world position to a screen-relative position before rendering.

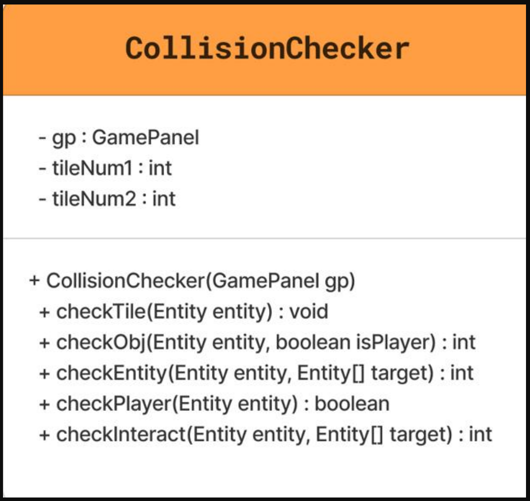
c) Formula:

- screenX = worldX - playerWorldX + playerScreenX
- screenY = worldY - playerWorldY + playerScreenY

with

- playerScreenX = screenWidth/2 – tileSize
- playerScreenY = screenHeight/2 - tileSize

2.6 Collision detection:


CollisionChecker
- gp : GamePanel
- tileNum1 : int
- tileNum2 : int
+ CollisionChecker(GamePanel gp)
+ checkTile(Entity entity) : void
+ checkObj(Entity entity, boolean isPlayer) : int
+ checkEntity(Entity entity, Entity[] target) : int
+ checkPlayer(Entity entity) : boolean
+ checkInteract(Entity entity, Entity[] target) : int

- Collision detection is a fundamental component in this game, responsible for determining when the player or other entities (such as monsters or NPCs) come into contact with obstacles, items, or other characters in the game world.

a) Purpose of collision detection:

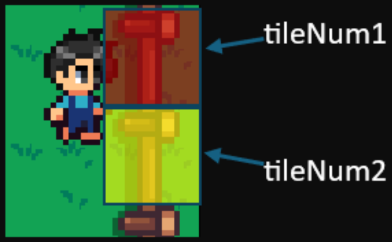
- The Collision Detector ensures that:

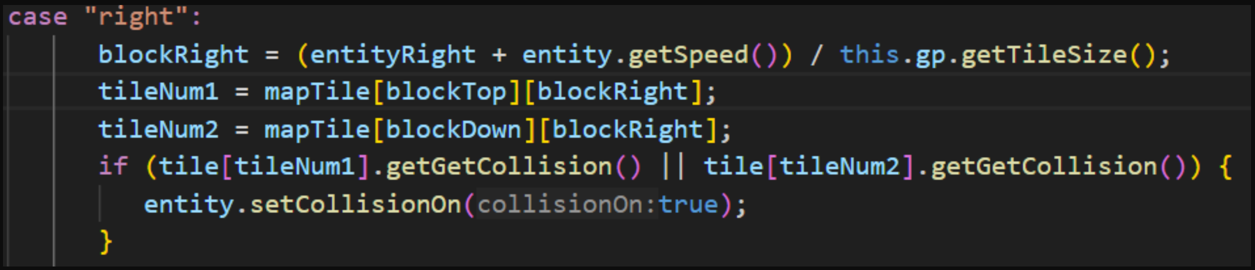
- The player cannot walk through solid objects (e.g., walls, rocks, or trees).
- Attacks or projectiles can accurately hit enemies.
- Interactions with objects (e.g., NPCs, chests, or potions) happen only when the player is close enough.
- Movement is restricted to valid, walkable areas.

b) How it works:

The game uses a rectangle-based detection system:

- Each entity has a hitbox, often defined as a Rectangle object.
- When movement is attempted, the game predicts the next position of the entity's hitbox based on current direction and speed. For example, if player is moving to the right, the hitbox on x-axis will be temporarily set to currentX + speed.
- It then checks if the new hitbox intersects with any tile or entity marked as collidable.





```
case "right":
    blockRight = (entityRight + entity.getSpeed()) / this.gp.getTileSize();
    tileNum1 = mapTile[blockTop][blockRight];
    tileNum2 = mapTile[blockDown][blockRight];
    if (tile[tileNum1].getGetCollision() || tile[tileNum2].getGetCollision()) {
        entity.setCollisionOn(collisionOn:true);
    }
```

c) Types of collisions:

There are typically four types of collision checks:

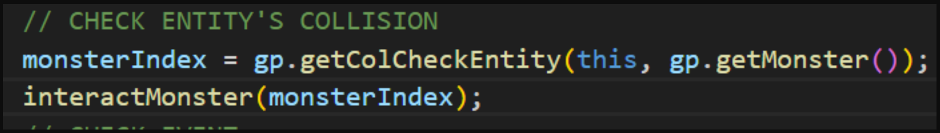
- Tile Collision: The detector checks whether the entity is about to move into a tile marked as solid (e.g., a fence or tree).
- Object Collision: Determines if the player is interacting with objects like items, treasure chests.
- Entity Collision: The detector checks whether the entity is about to collide with other entity.
- Entity Interaction: Used to check interaction between the player and other entities (e.g NPCs).

- Attack Collision: Special collisions for combat, e.g., checking if a sword swing or fireball overlaps an enemy's hitbox.

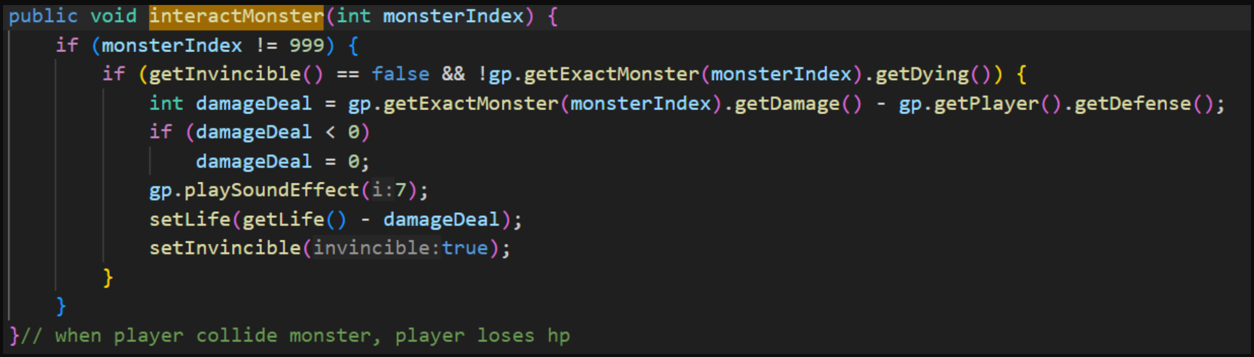
d) Entities behavior:

- Collision detection system not only detects overlap but also returns specific information about the collided entity. When the player collides with another entity, the system returns the index of the entity in its respective list.

- Once the entity is detected and identified, the game will use conditional logic to trigger the correct behavior. For instance, if player collides a monster, he will lose health.



```java
// CHECK ENTITY'S COLLISION
monsterIndex = gp.getColCheckEntity(this, gp.getMonster());
interactMonster(monsterIndex);
```



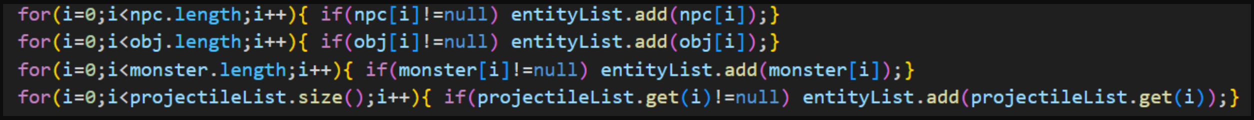
```java
public void interactMonster(int monsterIndex) {
    if (monsterIndex != 999) {
        if (getInvincible() == false && !gp.getExactMonster(monsterIndex).getDying()) {
            int damageDeal = gp.getExactMonster(monsterIndex).getDamage() - gp.getPlayer().getDefense();
            if (damageDeal < 0)
                damageDeal = 0;
            gp.playSoundEffect(i:7);
            setLife(getLife() - damageDeal);
            setInvincible(invincible:true);
        }
    }
}// when player collide monster, player loses hp
```

2.7 Render order

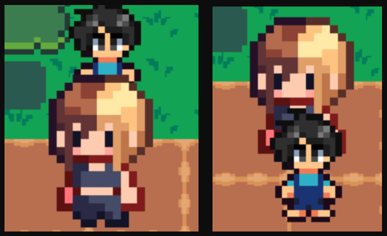
- Each frame is rendered in a strict order: base tiles(background layer) – overlay tiles – entities – projectile – UI components.

- To simulate depth and realistic visual layering in a 2D game, our game engine employs a Y-axis sorting technique for rendering all entities.

- All interactive entities—including objects, NPCs, Monsters, player character, are added into a single shared ArrayList during each frame. This unified list ensures that all drawable game entities are managed together for rendering purposes.

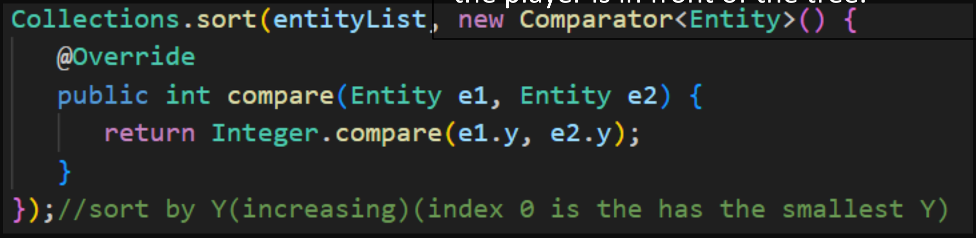


```java
for(i=0;i<npc.length;i++){ if(npc[i]!=null) entityList.add(npc[i]);}
for(i=0;i<obj.length;i++){ if(obj[i]!=null) entityList.add(obj[i]);}
for(i=0;i<monster.length;i++){ if(monster[i]!=null) entityList.add(monster[i]);}
for(i=0;i<projectileList.size();i++){ if(projectileList.get(i)!=null) entityList.add(projectileList.get(i));}
```



- Before rendering begins, the list is sorted using the Y-coordinate of each entity.

- After sorting, the game draws each entity from top to bottom, based on their Y position. For example, a tree rooted at Y = 100 will be drawn before a player standing at Y = 200, giving the illusion that the player is in front of the tree.



```
Collections.sort(entityList, new Comparator<Entity>() {
    @Override
    public int compare(Entity e1, Entity e2) {
        return Integer.compare(e1.y, e2.y);
    }
});//sort by Y(increasing)(index 0 is the has the smallest Y)
```

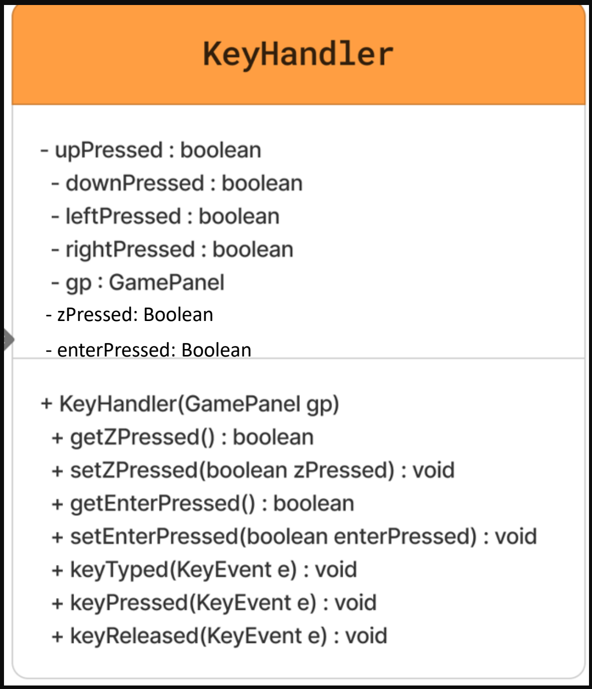
- The arrayList will be empty and reassigned after each loop

2.8 Key handler

- The game implements a Key Handler class responsible for detecting and managing keyboard input from player.

WASD	Move the player character (up/left/down/right)
TAB	Open inventory and display character stats
ENTER	- Confirm selection or use/equip item - Interact with nearby objects/NPCs the player.
R	Attack
F	Cast fireball skill
P	Pause or return to previous menu

- The input from player will behave differently in different game states. For example, in pause state, most keys are disabled excepts resume (P).



KeyHandler

- upPressed : boolean
- downPressed : boolean
- leftPressed : boolean
- rightPressed : boolean
- gp : GamePanel
- zPressed: Boolean
- enterPressed: Boolean

+ KeyHandler(GamePanel gp)
+ getZPressed() : boolean
+ setZPressed(boolean zPressed) : void
+ getEnterPressed() : boolean
+ setEnterPressed(boolean enterPressed) : void
+ keyTyped(KeyEvent e) : void
+ keyPressed(KeyEvent e) : void
+ keyReleased(KeyEvent e) : void

2.9 Hit cooldown

a) Purpose of hit cooldown:

- To prevent players or enemies from being damaged continuously, the game is equipped a hit cooldown system.

b) How it works:

-After an entity (such as the player or a monster) takes damage, the cooldown counter is activated immediately.

- With each iteration of the game loop, the counter is incremented by 1. During this cooldown period, the entity enters an invincible state, which temporarily prevents it from receiving any further damage.

- In our implementation, the cooldown period lasts for 60 frames, which is equivalent to 1 second at a frame rate of 60 FPS.

- Once the counter reaches this milestone, his invincibility is deactivated, and the entity becomes vulnerable to damage again.

c) Application in mana status

- To prevent the player from spamming skills, a mana system in applied in our game.

- Once the fireball is shooted, the mana will be decreased by 1, and if the mana bar is empty, the character can no longer shoots fireball until it reloads.

- The hit cooldown is also applied prevent player from spamming shooting fireball.

2.10 UI

- The game includes several core UI elements:

- Health bar: display the current health of player at the top-left corner of the screen.
- Mana bar: show available mana used for shooting fireball
- Inventory window (activated by pressing tab): display list of items currently held by the character.
- Character stats (activated by pressing tab): show detailed stats such as strength, defense, health, mana, level, EXP, EXP required for next level, current weapon, current shield.
- Dialogue boxes (activated by talking to NPCs): Text is displayed in a dialogue panel.
- Main menu (display at game launch): includes core options such as start game,quit game. If “new game” is selected, the game initializes; if the “quit” is selected, the application terminates via System.exit()
- Pause menu(triggered by pressing P): notice player that the game is paused.
- Game over screen: display 3 options for player (reset, respawn, quit)

2.11 Infinite rounds

- The infinite round system is designed to:

- Provide endless gameplay for survival-oriented players.
- Gradually increase difficulty over time to maintain tension.

-Each round in the game follows a consistent format:

- 10 enemies per round, consisting of:
    - 5 Slimes: Weaker monsters with long-range attacks.
    - 5 Skeletons: Stronger melee-type enemies.
- The player must defeat all enemies in the current round to proceed to the next.
- After completing each round all enemy stats are increased slightly.

- To do this, in each iteration of the main loop, we use a for loop to iterate through the ArrayList that holds all active monster entities. If all elements in the list are found to be null—indicating that all monsters in the current round have been defeated—the game automatically transitions the player to the next round.