INTERNATIONAL UNIVERSITY

VIETNAM NATIONAL UNIVERSITY, HCM CITY

School of Computer Science & Engineering

THE TREASURE HUNT PROJECT

Lecturor: Kieu Vu Thanh Tung

Course: Object – Oriented Programming Laboratory

Group members:

Trần Ngọc Đăng Khôi	ITCSIU21197
Nguyễn Trần Hoàng Hạ	ITITIU21127
Hà Văn Uyển Nhi	ITCSIU21095
Nguyễn Hoàng Quân	ITITIU21291

Link to project: koitran14/OOP.Project

CONTENTS

CHAPTER 1:
INTRODUCTION

In today's rapid evolution of technology industry, there is an increasing need for continuous learning and updating of knowledge. To facilitate the adoption of new ideas, Object-Oriented Programming has emerged as a robust paradigm for organizing projects that including code more reusability, modularity, maintainability, and scalability, to embrace new algorithms more easily and systematically.

Consequently, our team has made the choice to utilize Java.Swing as the primary library for coding and implementing Object-Oriented Programming (OOP) in our game project, "The Treasure Hunt". This decision was made in order to evaluate the viability of the principles inherent in this model. Furthermore, our goals encompassed enhancing our proficiency in collaborative projects and obtaining a more comprehensive comprehension of the game development process.

CHAPTER 2:

THE TREASURE HUNT

1. Content:

In this platform game, players take on the role of a pirate as the main character, navigating through a series of levels or stages filled with challenges and obstacles. The game is designed to be a two-dimensional side-scrolling environment, with the pirate moving from left to right or up and down, jumping over gaps, and dodging environmental hazards.

In this adventure, player will encounter a lot of enemies which can be found scuttling around the levels. The enemies have unique behaviors that players will need to learn in order to defeat them. They are aggressive and attack the player on sight. Players can use weapons, such as swords, to fight back against these enemies and progress through the game. Other challenges are cannons and traps, which are placed along the game worlds.

The game is designed to be engaging and challenging, with increasing levels of difficulty as players progress through the game. The use of pirates as the main character and crabs as enemies adds a unique flavor to the game, giving players a fun and immersive experience that will keep them coming back for more.

2. Rule:

The objective of this game is for the player to navigate through various levels, avoiding obstacles and enemies while collecting treasures along the way. The player has a limited level of health, which is lost if the player falls off the platform or gets hit by an enemy. The player can jump, move left or right, and perform other actions depending on the game mechanics. The game may also have different difficulty levels or modes to provide a challenge for players of varying skill levels. The ultimate goal of the game is to complete all levels and defeat all the enemies to win the game.

CHAPTER 3:
METHODOLOGY

1. Main library: Java.Swing

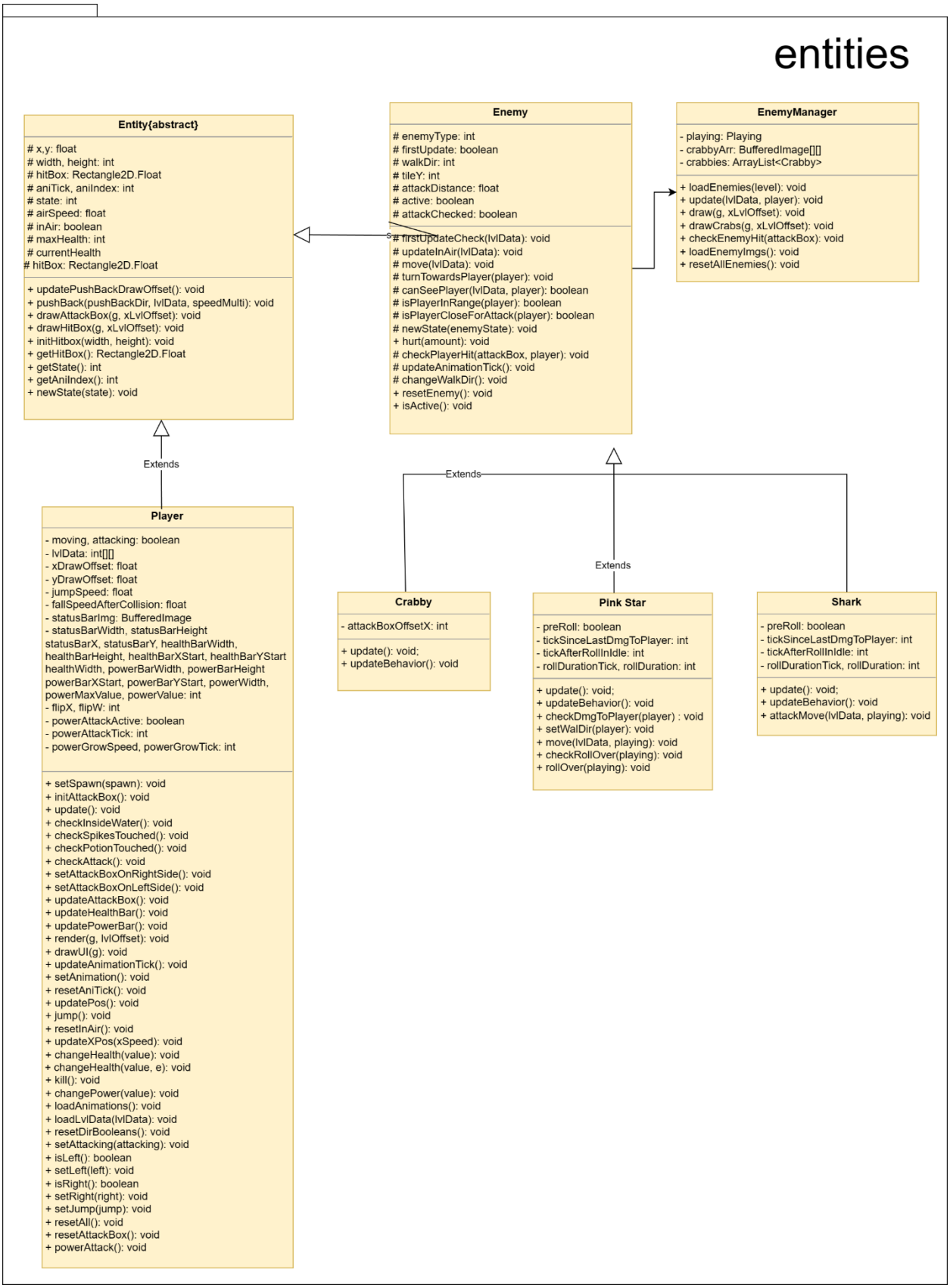
Java.Swing is a power and primary resource for Java developers to construct graphical user interfaces (GUIs). It offers an extensive range of components and features that empower developers to create engaging and visually captivating applications. This library allows developers to design and personalize buttons, menus, dialog boxes, and various graphical elements, thereby ensuring a smooth and immersive user experience. Additionally, it supports event-driven programming, enabling the development of responsive applications capable of efficiently managing user interactions.

Furthermore, our team made a deliberate choice to employ the Java.Swing library as the primary framework for our project, as opposed to alternatives like LibGDX [1]or LWJGL [2](Lightweight Java Game Library). The motivation behind this decision was to pursue a deeper comprehension of the underlying algorithms within the library"s functions. By adopting this approach, we aimed to engage in a meticulous examination of the nature of these functions, rather than relying solely on pre-existing built-in functionalities. This deliberate exploration of algorithms and their intricate workings is expected to yield a comprehensive understanding and serve as a valuable academic experience, contributing to our knowledge base for future related projects.

2. UML Diagrams:

[1] LibGDX: a cross-platform Java game development framework based on OpenGL (ES) that provides a well-tried and robust environment for rapid prototyping and fast iterations.
[2] LWJGL (Lightweight Java Game Library): a Java library that enables cross-platform access to popular native APIs useful in the development of graphics (OpenGL, Vulkan), audio (OpenAL) and parallel computing (OpenCL) applications.

entities


Entity{abstract}
x,y: float
width, height: int
hitBox: Rectangle2D.Float
aniTick, aniIndex: int
state: int
airSpeed: float
inAir: boolean
maxHealth: int
currentHealth
hitBox: Rectangle2D.Float
+ updatePushBackDrawOffset(): void
+ pushBack(pushBackDir, lvlData, speedMulti): void
+ drawAttackBox(g, xLvlOffset): void
+ drawHitBox(g, xLvlOffset): void
+ initHitbox(width, height): void
+ getHitBox(): Rectangle2D.Float
+ getState(): int
+ getAniIndex(): int
+ newState(state): void
Enemy
enemyType: int
firstUpdate: boolean
walkDir: int
tileY: int
attackDistance: float
active: boolean
attackChecked: boolean
firstUpdateCheck(lvlData): void
updateInAir(lvlData): void
move(lvlData): void
turnTowardsPlayer(player): void
canSeePlayer(lvlData, player): boolean
isPlayerInRange(player): boolean
isPlayerCloseForAttack(player): boolean
newState(enemyState): void
+ hurt(amount): void
# checkPlayerHit(attackBox, player): void
# updateAnimationTick(): void
# changeWalkDir(): void
+ resetEnemy(): void
+ isActive(): void
EnemyManager
- playing: Playing
- crabbyArr: BufferedImage[][]
- crabbies: ArrayList<Crabby>
+ loadEnemies(level): void
+ update(lvlData, player): void
+ draw(g, xLvlOffset): void
+ drawCrabs(g, xLvlOffset): void
+ checkEnemyHit(attackBox): void
+ loadEnemyImgs(): void
+ resetAllEnemies(): void
Extends
Player
- moving, attacking: boolean
- lvlData: int[][]
- xDrawOffset: float
- yDrawOffset: float
- jumpSpeed: float
- fallSpeedAfterCollision: float
- statusBarImg: BufferedImage
- statusBarWidth, statusBarHeight
statusBarX, statusBarY, healthBarWidth,
healthBarHeight, healthBarXStart, healthBarYStart
healthWidth, powerBarWidth, powerBarHeight
powerBarXStart, powerBarYStart, powerWidth,
powerMaxValue, powerValue: int
- flipX, flipW: int
- powerAttackActive: boolean
- powerAttackTick: int
- powerGrowSpeed, powerGrowTick: int
+ setSpawn(spawn): void
+ initAttackBox(): void
+ update(): void
+ checkInsideWater(): void
+ checkSpikesTouched(): void
+ checkPotionTouched(): void
+ checkAttack(): void
+ setAttackBoxOnRightSide(): void
+ setAttackBoxOnLeftSide(): void
+ updateAttackBox(): void
+ updateHealthBar(): void
+ updatePowerBar(): void
+ render(g, lvlOffset): void
+ drawUI(g): void
+ updateAnimationTick(): void
+ setAnimation(): void
+ resetAniTick(): void
+ updatePos(): void
+ jump(): void
+ resetInAir(): void
+ updateXPos(xSpeed): void
+ changeHealth(value): void
+ changeHealth(value, e): void
+ kill(): void
+ changePower(value): void
+ loadAnimations(): void
+ loadLvlData(lvlData): void
+ resetDirBooleans(): void
+ setAttacking(attacking): void
+ isLeft(): boolean
+ setLeft(left): void
+ isRight(): boolean
+ setRight(right): void
+ setJump(jump): void
+ resetAll(): void
+ resetAttackBox(): void
+ powerAttack(): void
Extends
Extends
Crabby
- attackBoxOffsetX: int
+ update(): void;
+ updateBehavior(): void
Pink Star
- preRoll: boolean
- tickSinceLastDmgToPlayer: int
- tickAfterRollInIdle: int
- rollDurationTick, rollDuration: int
+ update(): void;
+ updateBehavior(): void
+ checkDmgToPlayer(player) : void
+ setWalDir(player): void
+ move(lvlData, playing): void
+ checkRollOver(playing): void
+ rollOver(playing): void
Shark
- preRoll: boolean
- tickSinceLastDmgToPlayer: int
- tickAfterRollInIdle: int
- rollDurationTick, rollDuration: int
+ update(): void;
+ updateBehavior(): void
+ attackMove(lvlData, playing): void

utilz

Constants

+ GRAVITY: float
+ ANI_SPEED: int

+ FileType
+ Dialogue
+ Projectiles
+ ObjectConstants

FileType

+ MENU: String = "menu"
+ ENTITY: String = "entity"
+ OBJECT: String = "obj"

Dialogue

+ QUESTION = 0 {read only}
+ EXCLAMATION = 1 {read only}
+ DIALOGUE_WIDTH : int = (int) (14 * Game.SCALE) {read only}
+ DIALOGUE_HEIGHT: int = (int) (12* Game.SCALE) {read only}

+ GetSpriteAmount(type : int):int

HelpMethod

+ CanMoveHere(x: float, y:float, width:float, height:float, lvlData:int[][]): boolean
- IsSolid(x:float, y:float, lvlData: int[][]):boolean
+ IsProjectileHittingLevel(p: Projectile, lvlData: int[][]): boolean
+ IsEntityInWater(hitbox: Rectangle2D.Float, lvlData int[][]): boolean
- GetTileValue(xPos:float, yPos:float, lvlData: int[][]): int
+ IsTileSolid(xTile: int, yTile: int, lvlData: int[][]):boolean
+ GetEntityXPosNextToWall(hitbox: Rectangle2D.Float, xSpeed: float): float
+ GetEntityYPosUnderRoofOrAboveFloor (hitbox: Rectangle2D.Float, airSpeed: float): float
+ IsEntityOnFloor(hitbox: Rectangle2D.Float, lvlData int[][]): boolean
+ IsFloor (hitbox: Rectangle2D.Float, xSpeed: float, lvlData int[][]): boolean
+ CanCannonSeePlayer (lvlData: int[][], firstHitbox: Rectangle2D.Float, secondHitbox: Rectangle2D.Float, yTile: int): boolean
+ IsAllTilesClear (xStart: int, xEnd: int, y:int, lvlData: int[][]): boolean
+ IsAllTilesWalkable (xStart: int, xEnd: int, y:int, lvlData: int[][]): boolean
+ IsSightClear (lvlData: int[][], enemyBox: Rectangle2D.Float, playerBox: Rectangle2D.Float, yTile: int): boolean

Constants

+ PLAYER_ATLAS: String = "player_sprites.png" {Read only}
+ LEVEL_ATLAS: String = "outside_sprites.png" {Read only}
+ MENU_BUTTONS: String = "button_atlas.png" {Read only}
+ PAUSE_BACKGROUND: String = "pause_menu.png" {Read only}
+ SOUNDBUTTON: String = "sound_button.png" {Read only}
+ URM_BUTTON: String = "urm_buttons.png" {Read only}
+ VOLUME_BUTTONS: String = "volume_buttons.png" {Read only}
+ MENU_BACKGROUND_IMG: String = "background_menu.png" {Read only}
+ PLAYING_BG_IMG: String = "playing_bg_img.png" {Read only}
+ BIG_CLOUDS: String = "big_clouds.png" {Read only}
+ SMALL_CLOUDS: String = "small_clouds.png" {Read only}
+ CRABBY_SPRITE: String = "crabby_sprite.png" {Read only}
+ STATUS_BAR: String = "health_power_bar.png" {Read only}
+ COMPLETED_IMG: String = "completed_sprites.png" {Read only}
+ POTION_ATLAS: String = "potions_sprites.png" {Read only}
+ CONTAINER_ATLAS: String = "objects_sprites.png" {Read only}
+ TRAP_ATLAS: String = "trap_atlas.png" {Read only}
+ CANNON_ATLAS: String = "cannon_atlas.png" {Read only}
+ CANNON_BALL: String = "ball.png" {Read only}
+ DEATH_SCREEN: String = "death_screen.png" {Read only}
+ OPTIONS_MENU: String = "options_background.png" {Read only}
+ PINSTAR_ATLAS: String = "pinkstar_atlas.png" {Read only}
+ QUESTION_ATLAS: String = "question_atlas.png" {Read only}
+ EXCLAMATION_ATLAS: String = "exclamation_atlas.png" {Read only}
+ SHARK_ATLAS: String = "shark_atlas.png" {Read only}
+ GRASS_ATLAS: String = "grass_atlas.png" {Read only}
+ TREE_ONE_ATLAS: String = "tree_one_atlas.png" {Read only}
+ TREE_TWO_ATLAS: String = "tree_one_atlas.png" {Read only}
+ GAME_COMPLETED: String = "game_completed.png" {Read only}
+ RAIN_PARTICLE: String = "rain_particle.png" {Read only}
+ WATER_TOP: String = "water_atlas_animation.png" {Read only}
+ WATER_BOTTOM: String = "water.png" {Read only}
+ SHIP: String = "ship.png" {Read only}

+ GetSpriteAtlas (fileName: String, fileType: String): BufferedImage
+ GetAllLevels (): BufferedImage[]

Projectiles

+ CANNON_BALL_DEFAULT_WIDTH: int = 15 {read only}
+ CANNON_BALL_DEFAULT_WIDTH: int = 15 {read only}
+ CANNON_BALL_WIDTH: int = (int) (Game.SCALE * CANNON_BALL_DEFAULT_WIDTH) {read only}
+ CANNON_BALL_HEIGHT: int = (int) (Game.SCALE * CANNON_BALL_DEFAULT_HEIGHT) {read only}
+ SPEED : float = 0.5f * Game.SCALE; {read only}

ObjectConstants

+ GetSpriteAmount(object_type : int):int
+ GetTreeOffsetX(treeType: int) : int
+ GetTreeOffsetY(treeType: int) : int
+ GetTreeWidth(treeType: int) : int
+ GetTreeHeight(treeType: int) : int

+ RED_POTION: int = 0 {read only}
+ BLUE_POTION: int = 1 {read only}
+ BARREL: int = 2 {read only}
+ BOX : int = 3 {read only}
+ SPIKE: int = 4 {read only}
+ CANNON_LEFT: int = 5 {read only}
+ CANNON_RIGHT: int = 6 {read only}
+ TREE_ONE: int = 7 {read only}
+ TREE_TWO: int = 8 {read only}
+ TREE_THREE: int = 9 {read only}
+ RED_POTION_VALUE: int = 15 {read only}
+ BLUE_POTION_VALUE: int = 10 {read only}
+ CONTAINER_HEIGHT_DEFAULT: int = 30 {read only}
+ CONTAINER_WIDTH_DEFAULT: int = 40 {read only}
+ CONTAINER_HEIGHT: int = (int) (Game.SCALE * CONTAINER_HEIGHT_DEFAULT) {read only}
+ CONTAINER_WIDTH: int = (int) (Game.SCALE * CONTAINER_WIDTH_DEFAULT) {read only}
+ POTION_HEIGHT_DEFAULT: int = 16 {read only}
+ POTION_WIDTH_DEFAULT: int = 12 {read only}
+ POTION_HEIGHT: int = (int) (Game.SCALE * POTION_HEIGHT_DEFAULT) {read only}
+ POTION_WIDTH: int = (int) (Game.SCALE * POTION_WIDTH_DEFAULT) {read only}
+ SPIKE_HEIGHT_DEFAULT: int = 32 {read only}
+ SPIKE_WIDTH_DEFAULT: int = 32 {read only}
+ SPIKE_HEIGHT: int = (int) (Game.SCALE * SPIKE_HEIGHT_DEFAULT) {read only}
+ SPIKE_WIDTH: int = (int) (Game.SCALE * SPIKE_WIDTH_DEFAULT) {read only}
+ CANNON_HEIGHT_DEFAULT: int = 26 {read only}
+ CANNON_WIDTH_DEFAULT: int 40 {read only}
+ CANNON_HEIGHT: int = (int) (Game.SCALE * CANNON_HEIGHT_DEFAULT) {read only}
+ CANNON_WIDTH: int = (int) (Game.SCALE * CANNON_WIDTH_DEFAULT) {read only}

Gamestates

State

game: Game

+ isIn(e, mb): boolean
+ getGame(): Game
+ setGameState(state): void

<<enum>> Gamestate

PLAYING
MENU
OPTIONS
QUIT

Extends

GameOptions

- audioOptions: AudioOptions
- backgroundImg, optionsBackgroundImg: BufferedImage
- bgX, bgY, bgW, bgH: int
- menuB: UrmButton

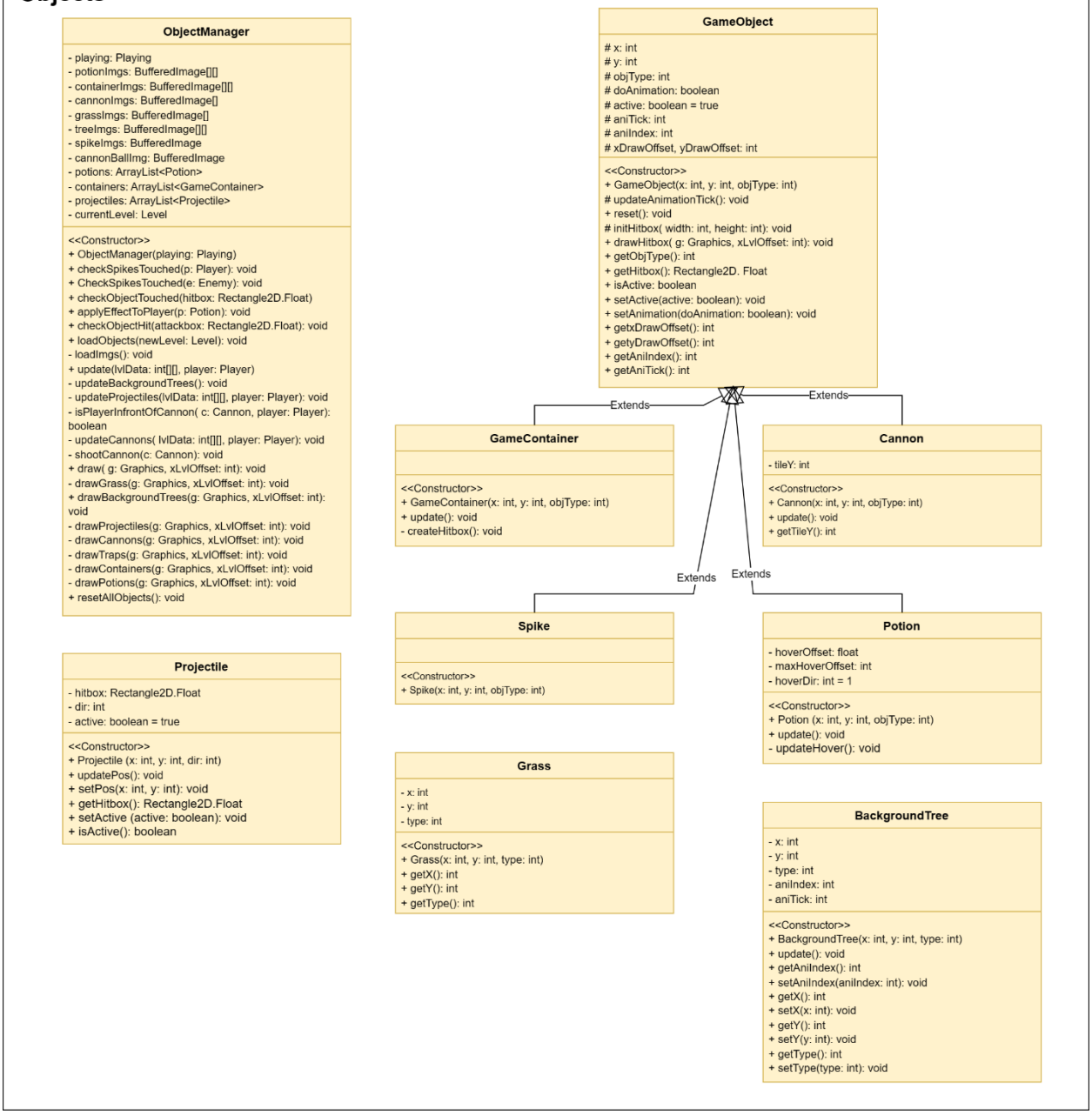
- loadButton(): void
- loadImgs(): void
- update(): void
- draw(g): void
- mouseDragged(e): void
- mousePressed(e): void
- mouseReleased(e): void
- mouseMoved(e): void
- keyPressed(e): void
- keyReleased(e): void
- mouseClicked(e): void
- isIn(e,b): boolean

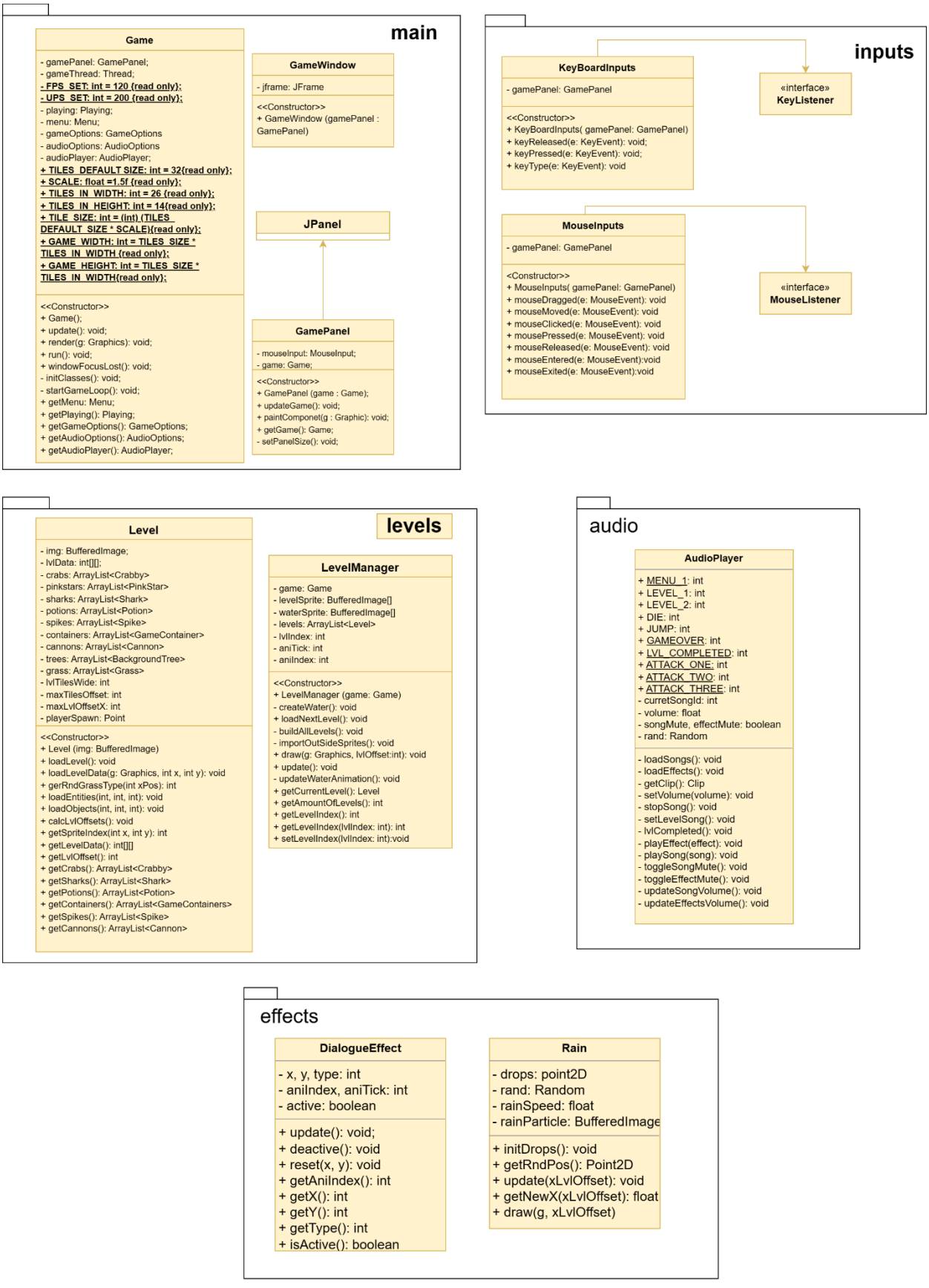
Menu

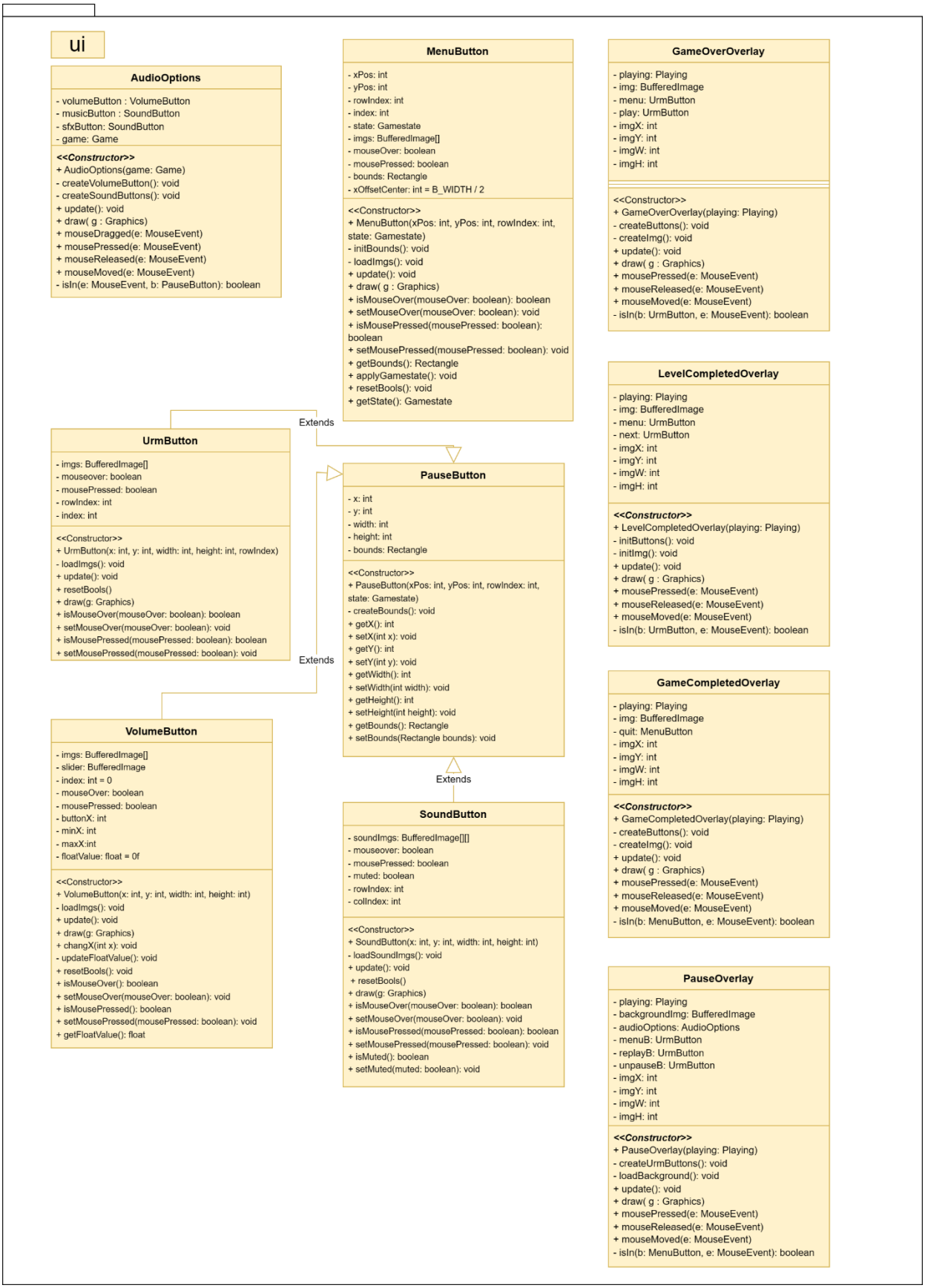
- buttons: MenuButton[]
- backgroundImg, backgroundImgPink: BufferedImage
- menuX, menuY, menuWidth, menuHeight: int

- loadBackground(): void
- loadButtons(): void
- update(): void
- draw(g): void
- mousePressed(e): void
- mouseReleased(e): void
- resetButtons(): void
- mouseMoved(e): void
- keyPressed(e): void
- keyReleased(e): void
- mouseClicked(e): void

Objects


ObjectManager
- playing: Playing
- potionImgs: BufferedImage[][]
- containerImgs: BufferedImage[][]
- cannonImgs: BufferedImage[]
- grassImgs: BufferedImage[]
- treeImgs: BufferedImage[][]
- spikeImgs: BufferedImage
- cannonBallImg: BufferedImage
- potions: ArrayList<Potion>
- containers: ArrayList<GameContainer>
- projectiles: ArrayList<Projectile>
- currentLevel: Level
<<Constructor>>
+ ObjectManager(playing: Playing)
+ checkSpikesTouched(p: Player): void
+ CheckSpikesTouched(e: Enemy): void
+ checkObjectTouched(hitbox: Rectangle2D.Float)
+ applyEffectToPlayer(p: Potion): void
+ checkObjectHit(attackbox: Rectangle2D.Float): void
+ loadObjects(newLevel: Level): void
- loadImgs(): void
+ update(lvlData: int[][], player: Player)
- updateBackgroundTrees(): void
- updateProjectiles(lvlData: int[][], player: Player): void
- isPlayerInfrontOfCannon(c: Cannon, player: Player): boolean
- updateCannons(lvlData: int[][], player: Player): void
- shootCannon(c: Cannon): void
+ draw(g: Graphics, xLvlOffset: int): void
- drawGrass(g: Graphics, xLvlOffset: int): void
+ drawBackgroundTrees(g: Graphics, xLvlOffset: int): void
- drawProjectiles(g: Graphics, xLvlOffset: int): void
- drawCannons(g: Graphics, xLvlOffset: int): void
- drawTraps(g: Graphics, xLvlOffset: int): void
- drawContainers(g: Graphics, xLvlOffset: int): void
- drawPotions(g: Graphics, xLvlOffset: int): void
+ resetAllObjects(): void
Projectile
- hitbox: Rectangle2D.Float
- dir: int
- active: boolean = true
<<Constructor>>
+ Projectile (x: int, y: int, dir: int)
+ updatePos(): void
+ setPos(x: int, y: int): void
+ getHitbox(): Rectangle2D.Float
+ setActive (active: boolean): void
+ isActive(): boolean
GameObject
x: int
y: int
objType: int
doAnimation: boolean
active: boolean = true
aniTick: int
aniIndex: int
xDrawOffset, yDrawOffset: int
<<Constructor>>
+ GameObject(x: int, y: int, objType: int)
# updateAnimationTick(): void
+ reset(): void
# initHitbox(width: int, height: int): void
+ drawHitbox(g: Graphics, xLvlOffset: int): void
+ getObjType(): int
+ getHitbox(): Rectangle2D. Float
+ isActive: boolean
+ setActive(active: boolean): void
+ setAnimation(doAnimation: boolean): void
+ getxDrawOffset(): int
+ getyDrawOffset(): int
+ getAniIndex(): int
+ getAniTick(): int
Extends
Extends
GameContainer
<<Constructor>>
+ GameContainer(x: int, y: int, objType: int)
+ update(): void
- createHitbox(): void
Cannon
- tileY: int
<<Constructor>>
+ Cannon(x: int, y: int, objType: int)
+ update(): void
+ getTileY(): int
Extends
Extends
Spike
<<Constructor>>
+ Spike(x: int, y: int, objType: int)
Potion
- hoverOffset: float
- maxHoverOffset: int
- hoverDir: int = 1
<<Constructor>>
+ Potion (x: int, y: int, objType: int)
+ update(): void
- updateHover(): void
Grass
- x: int
- y: int
- type: int
<<Constructor>>
+ Grass(x: int, y: int, type: int)
+ getX(): int
+ getY(): int
+ getType(): int
BackgroundTree
- x: int
- y: int
- type: int
- aniIndex: int
- aniTick: int
<<Constructor>>
+ BackgroundTree(x: int, y: int, type: int)
+ update(): void
+ getAniIndex(): int
+ setAniIndex(aniIndex: int): void
+ getX(): int
+ setX(x: int): void
+ getY(): int
+ setY(y: int): void
+ getType(): int
+ setType(type: int): void


main
Game
- gamePanel: GamePanel;
- gameThread: Thread;
- FPS_SET: int = 120 {read only};
- UPS_SET: int = 200 {read only};
- playing: Playing;
- menu: Menu;
- gameOptions: GameOptions
- audioOptions: AudioOptions
- audioPlayer: AudioPlayer;
+ TILES_DEFAULT SIZE: int = 32{read only};
+ SCALE: float =1.5f {read only};
+ TILES_IN_WIDTH: int = 26 {read only};
+ TILES_IN_HEIGHT: int = 14{read only};
+ TILE_SIZE: int = (int) (TILES_DEFAULT_SIZE * SCALE){read only};
+ GAME_WIDTH: int = TILES_SIZE * TILES_IN_WIDTH {read only};
+ GAME_HEIGHT: int = TILES_SIZE * TILES_IN_WIDTH{read only};
<<Constructor>>
+ Game();
+ update(): void;
+ render(g: Graphics): void;
+ run(): void;
+ windowFocusLost(): void;
- initClasses(): void;
- startGameLoop(): void;
+ getMenu: Menu;
+ getPlaying(): Playing;
+ getGameOptions(): GameOptions;
+ getAudioOptions(): AudioOptions;
+ getAudioPlayer(): AudioPlayer;
GameWindow
- jframe: JFrame
<<Constructor>>
+ GameWindow (gamePanel : GamePanel)
JPanel
GamePanel
- mouseInput: MouseInput;
- game: Game;
<<Constructor>>
+ GamePanel (game : Game);
+ updateGame(): void;
+ paintComponet(g : Graphic): void;
+ getGame(): Game;
- setPanelSize(): void;
inputs
KeyBoardInputs
- gamePanel: GamePanel
<<Constructor>>
+ KeyBoardInputs(gamePanel: GamePanel)
+ keyReleased(e: KeyEvent): void;
+ keyPressed(e: KeyEvent): void;
+ keyType(e: KeyEvent): void
«interface»
KeyListener
MouseInputs
- gamePanel: GamePanel
<Constructor>>
+ MouseInputs(gamePanel: GamePanel)
+ mouseDragged(e: MouseEvent): void
+ mouseMoved(e: MouseEvent): void
+ mouseClicked(e: MouseEvent): void
+ mousePressed(e: MouseEvent): void
+ mouseReleased(e: MouseEvent): void
+ mouseEntered(e: MouseEvent):void
+ mouseExited(e: MouseEvent):void
«interface»
MouseListener
levels
Level
- img: BufferedImage;
- lvlData: int[][];
- crabs: ArrayList<Crabby>
- pinkstars: ArrayList<PinkStar>
- sharks: ArrayList<Shark>
- potions: ArrayList<Potion>
- spikes: ArrayList<Spike>
- containers: ArrayList<GameContainer>
- cannons: ArrayList<Cannon>
- trees: ArrayList<BackgroundTree>
- grass: ArrayList<Grass>
- lvlTilesWide: int
- maxTilesOffset: int
- maxLvlOffsetX: int
- playerSpawn: Point
<<Constructor>>
+ Level (img: BufferedImage)
+ loadLevel(): void
+ loadLevelData(g: Graphics, int x, int y): void
+ gerRndGrassType(int xPos): int
+ loadEntities(int, int, int): void
+ loadObjects(int, int, int): void
+ calcLvlOffsets(): void
+ getSpriteIndex(int x, int y): int
+ getLevelData(): int[][]
+ getLvlOffset(): int
+ getCrabs(): ArrayList<Crabby>
+ getSharks(): ArrayList<Shark>
+ getPotions(): ArrayList<Potion>
+ getContainers(): ArrayList<GameContainers>
+ getSpikes(): ArrayList<Spike>
+ getCannons(): ArrayList<Cannon>
LevelManager
- game: Game
- levelSprite: BufferedImage[]
- waterSprite: BufferedImage[]
- levels: ArrayList<Level>
- lvlIndex: int
- aniTick: int
- aniIndex: int
<<Constructor>>
+ LevelManager (game: Game)
- createWater(): void
+ loadNextLevel(): void
- buildAllLevels(): void
- importOutSideSprites(): void
+ draw(g: Graphics, lvlOffset:int): void
+ update(): void
- updateWaterAnimation(): void
+ getCurrentLevel(): Level
+ getAmountOfLevels(): int
+ getLevelIndex(): int
+ getLevelIndex(lvlIndex: int): int
+ setLevelIndex(lvlIndex: int):void
audio
AudioPlayer
+ MENU_1: int
+ LEVEL_1: int
+ LEVEL_2: int
+ DIE: int
+ JUMP: int
+ GAMEOVER: int
+ LVL_COMPLETED: int
+ ATTACK_ONE: int
+ ATTACK_TWO: int
+ ATTACK_THREE: int
- curretSongId: int
- volume: float
- songMute, effectMute: boolean
- rand: Random
- loadSongs(): void
- loadEffects(): void
- getClip(): Clip
- setVolume(volume): void
- stopSong(): void
- setLevelSong(): void
- lvlCompleted(): void
- playEffect(effect): void
- playSong(song): void
- toggleSongMute(): void
- toggleEffectMute(): void
- updateSongVolume(): void
- updateEffectsVolume(): void
effects
DialogueEffect
- x, y, type: int
- aniIndex, aniTick: int
- active: boolean
+ update(): void;
+ deactive(): void
+ reset(x, y): void
+ getAniIndex(): int
+ getX(): int
+ getY(): int
+ getType(): int
+ isActive(): boolean
Rain
- drops: point2D
- rand: Random
- rainSpeed: float
- rainParticle: BufferedImage
+ initDrops(): void
+ getRndPos(): Point2D
+ update(xLvlOffset): void
+ getNewX(xLvlOffset): float
+ draw(g, xLvlOffset)

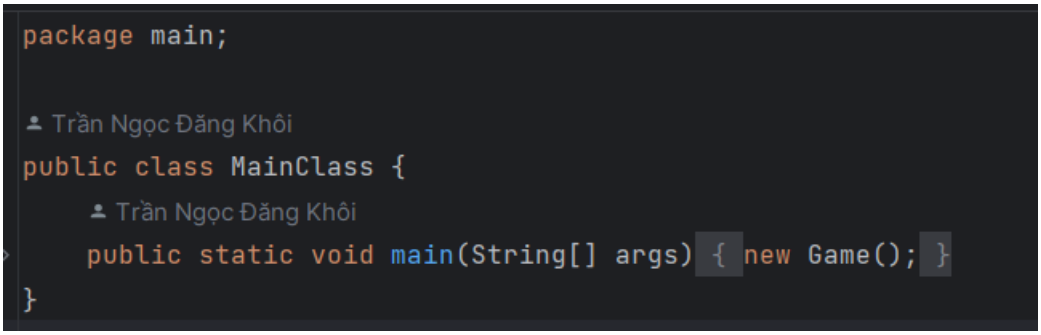

ui
AudioOptions
- volumeButton : VolumeButton
- musicButton : SoundButton
- sfxButton: SoundButton
- game: Game
<<Constructor>>
+ AudioOptions(game: Game)
- createVolumeButton(): void
- createSoundButtons(): void
+ update(): void
+ draw(g : Graphics)
+ mouseDragged(e: MouseEvent)
+ mousePressed(e: MouseEvent)
+ mouseReleased(e: MouseEvent)
+ mouseMoved(e: MouseEvent)
- isIn(e: MouseEvent, b: PauseButton): boolean
MenuButton
- xPos: int
- yPos: int
- rowIndex: int
- index: int
- state: Gamestate
- imgs: BufferedImage[]
- mouseOver: boolean
- mousePressed: boolean
- bounds: Rectangle
- xOffsetCenter: int = B_WIDTH / 2
<<Constructor>>
+ MenuButton(xPos: int, yPos: int, rowIndex: int, state: Gamestate)
- initBounds(): void
- loadImgs(): void
+ update(): void
+ draw(g : Graphics)
+ isMouseOver(mouseOver: boolean): boolean
+ setMouseOver(mouseOver: boolean): void
+ isMousePressed(mousePressed: boolean): boolean
+ setMousePressed(mousePressed: boolean): void
+ getBounds(): Rectangle
+ applyGamestate(): void
+ resetBools(): void
+ getState(): Gamestate
GameOverOverlay
- playing: Playing
- img: BufferedImage
- menu: UrmButton
- play: UrmButton
- imgX: int
- imgY: int
- imgW: int
- imgH: int
<<Constructor>>
+ GameOverOverlay(playing: Playing)
- createButtons(): void
- createImg(): void
+ update(): void
+ draw(g : Graphics)
+ mousePressed(e: MouseEvent)
+ mouseReleased(e: MouseEvent)
+ mouseMoved(e: MouseEvent)
- isIn(b: UrmButton, e: MouseEvent): boolean
Extends
UrmButton
- imgs: BufferedImage[]
- mouseover: boolean
- mousePressed: boolean
- rowIndex: int
- index: int
<<Constructor>>
+ UrmButton(x: int, y: int, width: int, height: int, rowIndex)
- loadImgs(): void
+ update(): void
+ resetBools()
+ draw(g: Graphics)
+ isMouseOver(mouseOver: boolean): boolean
+ setMouseOver(mouseOver: boolean): void
+ isMousePressed(mousePressed: boolean): boolean
+ setMousePressed(mousePressed: boolean): void
PauseButton
- x: int
- y: int
- width: int
- height: int
- bounds: Rectangle
<<Constructor>>
+ PauseButton(xPos: int, yPos: int, rowIndex: int, state: Gamestate)
- createBounds(): void
+ getX(): int
+ setX(int x): void
+ getY(): int
+ setY(int y): void
+ getWidth(): int
+ setWidth(int width): void
+ getHeight(): int
+ setHeight(int height): void
+ getBounds(): Rectangle
+ setBounds(Rectangle bounds): void
LevelCompletedOverlay
- playing: Playing
- img: BufferedImage
- menu: UrmButton
- next: UrmButton
- imgX: int
- imgY: int
- imgW: int
- imgH: int
<<Constructor>>
+ LevelCompletedOverlay(playing: Playing)
- initButtons(): void
- initImg(): void
+ update(): void
+ draw(g : Graphics)
+ mousePressed(e: MouseEvent)
+ mouseReleased(e: MouseEvent)
+ mouseMoved(e: MouseEvent)
- isIn(b: UrmButton, e: MouseEvent): boolean
Extends
VolumeButton
- imgs: BufferedImage[]
- slider: BufferedImage
- index: int = 0
- mouseOver: boolean
- mousePressed: boolean
- buttonX: int
- minX: int
- maxX:int
- floatValue: float = 0f
<<Constructor>>
+ VolumeButton(x: int, y: int, width: int, height: int)
- loadImgs(): void
+ update(): void
+ draw(g: Graphics)
+ changX(int x): void
- updateFloatValue(): void
+ resetBools(): void
+ isMouseOver(): boolean
+ setMouseOver(mouseOver: boolean): void
+ isMousePressed(): boolean
+ setMousePressed(mousePressed: boolean): void
+ getFloatValue(): float
Extends
SoundButton
- soundImgs: BufferedImage[][]
- mouseover: boolean
- mousePressed: boolean
- muted: boolean
- rowIndex: int
- colIndex: int
<<Constructor>>
+ SoundButton(x: int, y: int, width: int, height: int)
- loadSoundImgs(): void
+ update(): void
+ resetBools()
+ draw(g: Graphics)
+ isMouseOver(mouseOver: boolean): boolean
+ setMouseOver(mouseOver: boolean): void
+ isMousePressed(mousePressed: boolean): boolean
+ setMousePressed(mousePressed: boolean): void
+ isMuted(): boolean
+ setMuted(muted: boolean): void
GameCompletedOverlay
- playing: Playing
- img: BufferedImage
- quit: MenuButton
- imgX: int
- imgY: int
- imgW: int
- imgH: int
<<Constructor>>
+ GameCompletedOverlay(playing: Playing)
- createButtons(): void
- createImg(): void
+ update(): void
+ draw(g : Graphics)
+ mousePressed(e: MouseEvent)
+ mouseReleased(e: MouseEvent)
+ mouseMoved(e: MouseEvent)
- isIn(b: MenuButton, e: MouseEvent): boolean
PauseOverlay
- playing: Playing
- backgroundImg: BufferedImage
- audioOptions: AudioOptions
- menuB: UrmButton
- replayB: UrmButton
- unpauseB: UrmButton
- imgX: int
- imgY: int
- imgW: int
- imgH: int
<<Constructor>>
+ PauseOverlay(playing: Playing)
- createUrmButtons(): void
- loadBackground(): void
+ update(): void
+ draw(g : Graphics)
+ mousePressed(e: MouseEvent)
+ mouseReleased(e: MouseEvent)
+ mouseMoved(e: MouseEvent)
- isIn(b: MenuButton, e: MouseEvent): boolean

a. Main package:

The purpose of the **Main package** is to serve as the central coordination and entry point for the game project. It encompasses classes that handle the initialization of game components, the creation of the game window, game panel and the management of game states. The package plays a crucial role in setting up the foundation for the game and ensuring its proper execution.

i) Main class:

This class contains the main method, which has a role as the entry point for the entire program. Its main purpose is to instantiate the Game class, initiating the game and triggering its execution.



The Main Class

ii) GameWindow class:

The "GameWindow" class is responsible for creating the game window using Java.Swing library. Its purpose is to configure the window's properties, such as the title, close operation, size, and visibility. It also adds the game panel, which is responsible for rendering the game, to the window. Additionally, it includes a "WindowFocusListener [3]" to handle events related to the focus of the game window.

[3] *WindowFocusListener:* The listener interface for monitoring and responding to change in the focus state of a window or frame in a graphical user interface (GUI) application. It is an interface in Java's AWT (Abstract Window Toolkit) and Swing libraries.

```
public GameWindow(GamePanel gamePanel) {

    jframe = new JFrame( title: "The Treasure Hunt");
    jframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jframe.add(gamePanel);
    jframe.setResizable(false);
    jframe.pack();
    jframe.setLocationRelativeTo(null);
    jframe.setVisible(true);
```

The constructor GameWindow(GamePanel gamePanel)

iii) GamePanel class:

The GamePanel class is responsible for creating and managing the panel used to render the game. It extends the "JPanel [4]" class from the Swing library, allowing it to provide a graphical user interface component. The class contains a "Game" object and "MouseInputs" object, which handle the game logic and mouse inputs, respectively.

```
public class GamePanel extends JPanel {

    3 usages
    private MouseInputs mouseInputs;
    3 usages
    private Game game;
```

The GamePanel class

In the constructor, a "MouseInputs" instance is created, passing the current "GamePanel" object as a parameter. The "GamePanel" also receives a reference to the "Game" object. The "setPanelSize()" method is called to set the preferred size of the panel to match the dimensions of the game defined in the "Game" class. Key and mouse listeners are added to the panel to capture user input.

[4] *JPanel:* a class of Java.Swing package, which is extended from the Java AWT package. It includes and can combine various components such as labels, buttons and other panels also.

```
1 usage   Trần Ngọc Đăng Khôi
public GamePanel(Game game) {
    mouseInputs = new MouseInputs( gamePanel: this);
    this.game = game;
    setPanelSize();
    addKeyListener(new KeyboardInputs( gamePanel: this));
    addMouseListener(mouseInputs);
    addMouseMotionListener(mouseInputs);
}
```

The GamePanel(Game game) constructor

```
private void setPanelSize() {
    Dimension size = new Dimension(GAME_WIDTH, GAME_HEIGHT);
    setPreferredSize(size);
}
```

The setPanelSize() method

Moreover, this class also include he **"updateGame()"** method but it is currently empty and serves as a placeholder for updating the game state. It can be expanded with additional logic to update the game based on user input or other factors.

```
public void updateGame() {
}
```

The updateGame() method

The **"paintComponent()"** method is overridden from the "JPanel" class and is responsible for rendering the game graphics. It calls the "render()" method of the Game object, passing the "Graphics[5]" object as a parameter. This allows the "Game" class to handle the rendering of the game elements.

[5] *Graphics object:* an abstract class provided by Java AWT and used to draw or paint on the components. It also consists of various fields which hold information like components to be painted, font, color, XOR mod, etc., and methods to draw various shapes on the GUI components.

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    game.render(g);
}
```

The pain Component(Graphics g) method

The **"getGame()"** method is a getter method that returns the "Game" object associated with the "GamePanel" and created to ensure the security of data.

```
public Game getGame() { return game; }
```

The getGame() method

iv) Game class:

Implemented from the "Runnable [6]" interface, the "**Game" class** serves as the core component of the game project. Its purpose is to implement the game loop that continuously updates the game state, renders the game graphics, and tracks the FPS and UPS. It also manages different game states and provides methods to access game-related objects and options.

[6] *The "Runnable" interface:* a functional interface in Java that represents a task that can be excuted concurrently in a separate thread, defined in the "java.lang" package.

```
public class Game implements Runnable {

    4 usages
    private GamePanel gamePanel;
    2 usages
    private Thread gameThread;
    1 usage
    private final int FPS_SET = 120;
    1 usage
    private final int UPS_SET = 200;

    5 usages
    private Playing playing;
    4 usages
    private Menu menu;
    4 usages
    private GameOptions gameOptions;
    2 usages
    private AudioOptions audioOptions;
    2 usages
    private AudioPlayer audioPlayer;
```

Some attributes of the Game class

The **"Game()" constructor** is included in this class to initialize the game by creating instances of the necessary classes, setting up the game panel, creating a game window, and starting the gameloop by calling the "startGameLoop()" method.

```
public Game() {
    System.out.println("size: " + GAME_WIDTH + " : " + GAME_HEIGHT);
    initClasses();
    gamePanel = new GamePanel(this);
    new GameWindow(gamePanel);
    gamePanel.requestFocusInWindow();
    startGameLoop();
}
```

The Game constructor

Moreover, **"initClasses()"** method that loaded in the constructor created to initialize all the classes, which include audio (audio player and audio options), menu of the game, set up the playing statement and the game options of the game.

```
private void initClasses() {
    audioOptions = new AudioOptions( game: this);
    audioPlayer = new AudioPlayer();
    menu = new Menu( game: this);
    playing = new Playing( game: this);
    gameOptions = new GameOptions(this);
}
```

The initClasses method

On the other hands, **startGameLoop()** method is loaded in order to create a new thread [7]("gameThread**")** and start it, which triggers the execution of the "run()" method.

```
private void startGameLoop() {
    gameThread = new Thread( task: this);
    gameThread.start();
}
```

The startGameLoop method

Additionally, to update the game state based on the current "Gamestate.state", we create the **update()** method. It uses a switch statement to determine the appropriate update method to call for each game state.

```
public void update() {
    switch (Gamestate.state) {
        case MENU -> menu.update();
        case PLAYING -> playing.update();
        case OPTIONS -> gameOptions.update();
        case QUIT -> System.exit( status: 0);
    }
}
```

The update method

[7] Thread: pre-defined classes in Java that is available in the java.package. It allows multiple tasks to run concurrently, enabling to perform multiple operations simultaneously or asynchronouly.

The render() methods handles the rendering of game graphics using the provided “Graphics” object. It also uses a switch statement to determine which drawing method to call based on the current “Gamestate.state”.

```
1 usage   Trần Ngọc Đăng Khôi
public void render(Graphics g) {
    switch (Gamestate.state) {
        case MENU -> menu.draw(g);
        case PLAYING -> playing.draw(g);
        case OPTIONS -> gameOptions.draw(g);
    }
}
```

The render method

The core of the game loop is **run()** method. It calculates the time per frame and time per update based on the desired frames of per second (FPS) and updates the per second (UPS). Inside the loop, it tracks the time passed since the previous update, updates the game state if necessary, renders the game graphics, and calculates the FPS and UPS. The loop runs indefinitely until the program is terminated.

```
//FPS + UPS
 Trần Ngọc Đăng Khôi *
@Override
public void run() {
    double timePerFrame = 1000000000.0 / FPS_SET;
    double timePerUpdate = 1000000000.0 / UPS_SET;

    long previousTime = System.nanoTime();

    int frames = 0;
    int updates = 0;
    long lastCheck = System.currentTimeMillis();

    double deltaU = 0;
    double deltaF = 0;
```

```
    while (true) {
        long currentTime = System.nanoTime();

        deltaU += (currentTime - previousTime) / timePerUpdate;
        deltaF += (currentTime - previousTime) / timePerFrame;
        previousTime = currentTime;

        if (deltaU >= 1) {
            update();
            updates++;
            deltaU--;
        }
        if (deltaF >= 1) {
            gamePanel.repaint();
            frames++;
            deltaF--;
        }
            if (System.currentTimeMillis() - lastCheck >= 1000) {
                lastCheck = System.currentTimeMillis();
                System.out.println("FPS: " + frames + " | UPS: " + updates);
                frames = 0;
                updates = 0;
            }
    }
}
```

The run method

Next, the "**windowFocusLost()**" method is called when the game window loses focus. It checks if the game state is "PLAYING" and resets the directional booleans of the player.

```
1 usage   Trần Ngọc Đăng Khôi
public void windowFocusLost() {
    if (Gamestate.state == Gamestate.PLAYING)
        playing.getPlayer().resetDirBooleans();
}
```

The windowFocusLost method

Finally, we provides getter methods to access various game-related objects and options.

```
Trần Ngọc Đăng Khôi
public Menu getMenu() { return menu; }

Trần Ngọc Đăng Khôi
public Playing getPlaying() { return playing; }

5 usages   Trần Ngọc Đăng Khôi
public GameOptions getGameOptions() { return gameOptions; }

2 usages   Trần Ngọc Đăng Khôi
public AudioOptions getAudioOptions() { return audioOptions; }

14 usages   Trần Ngọc Đăng Khôi
public AudioPlayer getAudioPlayer() { return audioPlayer; }
```

The getter method

b. Levels package:

To load and store information about game levels and mage all related elements, we create **"levels"** package.

i) "Level" class:

The **"Level"** class within the package is responsible for loading and storing information about a game level. It contains various instance variables to store

different elements of a game level, such as the level image (“img”), level data (“lvlData”), list of entities (e.g., “crabs”, “pinkstars”, “sharks”), objects (e.g., “potions”, “spikes”, “cannons”), and environmental elements (e.g., “trees”, “grass”).

```
5 usages
private BufferedImage img;
5 usages
private int[][] lvlData;
2 usages
private ArrayList<Crabby> crabs = new ArrayList<>();
2 usages
private ArrayList<Pinkstar> pinkstars = new ArrayList<>();
2 usages
private ArrayList<Shark> sharks = new ArrayList<>();
2 usages
private ArrayList<Potion> potions = new ArrayList<>();
2 usages
private ArrayList<Spike> spikes = new ArrayList<>();
2 usages
private ArrayList<GameContainer> containers = new ArrayList<>();
2 usages
private ArrayList<Cannon> cannons = new ArrayList<>();
2 usages
private ArrayList<BackgroundTree> trees = new ArrayList<>();
2 usages
private ArrayList<Grass> grass = new ArrayList<>();
```

The attributes of “Level” Class

Moreover, there are some variables also created to calculate the limitation of the screen for rendering the level and determining the edge-reaching, such as “lvlTilesWide”, “maxTilesOffset” and “maxLvlOffsetX”. Besides, “playerSpawn” variable stores the spawn point for the player in the level.

```
2 usages
private int lvlTilesWide;
2 usages
private int maxTilesOffset;
2 usages
private int maxLvlOffsetX;
2 usages
private Point playerSpawn;
```

The attributes that support for calculating

The **Level construtor** is created to take an image parameter representing the level image. It initializes the instance variables and then calls the "loadLevel()" and "calcLvlOffsets()" methods to populate the level data and calculate the level offsets.

```
1 usage  Trần Ngọc Đăng Khôi
public Level(BufferedImage img) {
    this.img = img;
    lvlData = new int[img.getHeight()][img.getWidth()];
    loadLevel();
    calcLvlOffsets();
}
```

The constructor Level

The **"calcLvlOffsets()"** method calculates the level offsets used for rendering and determine the edge – reaching of the screen by multiplying the image width ("lvlTileWide") with the number of tiles in the game ("maxTilesOffset").

```
private void calcLvlOffsets() {
    lvlTilesWide = img.getWidth();
    maxTilesOffset = lvlTilesWide - Game.TILES_IN_WIDTH;
    maxLvlOffsetX = Game.TILES_SIZE * maxTilesOffset;
}
```

The calcLvlOffsets method

The purpose of calculating these offsets is to determine the range of rendering for the level and to restric the player's movement within the level boundaries. It ensures that the level is displayed within the screen limits and prevents the player from moving beyond the visible area.

To process the level image, we create the "**loadLevel()"** method to loop through its color based on the RGB [8] values and extract relevant type of elements to populate the various lists and level data.

```
private void loadLevel() {
    for (int y = 0; y < img.getHeight(); y++)
        for (int x = 0; x < img.getWidth(); x++) {
            Color c = new Color(img.getRGB(x, y));
            int red = c.getRed();
            int green = c.getGreen();
            int blue = c.getBlue();

            loadLevelData(red, x, y);
            loadEntities(green, x, y);
            loadObjects(blue, x, y);
        }
}
```

The loadLevel method

In detail, the **loadLevelData(), loadEntities(),** and **loadObjects()** methods handle the extraction of specific data based on color values and populate the appropriate lists.

*** loadLevelData() method:**

This method processes the initial red values ("redValue") in the level image, determines the type of tile or special condition they represent, transformes into the code and updates the "lvlData" array:

```
private void loadLevelData(int redValue, int x, int y) {
    if (redValue >= 50)
        lvlData[y][x] = 0;
    else
        lvlData[y][x] = redValue;
    switch (redValue) {
        case 0, 1, 2, 3, 30, 31, 33, 34, 35, 36, 37, 38, 39 ->
                grass.add(new Grass((int) (x * Game.TILES_SIZE), y: (int) (y * Game.TILES_SIZE) - Game.TILES_SIZE, getRndGrassType(x)));
    }
}
```

The loadLevelData method

[8] *RGB (Red Green Blue)***:** a part of Color class and includes the red, blue, and green components of a color are each represented by an interger in the range 0 -255.

To do that, the method has some checks on the red value (“redValue”): if the red value is greater than or equal to 50, it means that it represents a special condition in the level, and corresponding tile in the “lvlData” array is set to 0. Otherwise, it represents a grass tile and the “lvlData” array is set to the red value. Addtionally, a “Grass” object is also initialized with the position based on the x and y coordinates of the pixel and the grass type, which randomly chosen by the **getRndGrassType(x)** method.

```
private int getRndGrassType(int xPos) { return xPos % 2; }
```

The getRndGrassType method

*** loadEntities () method:**

This class has responsibily for loading and creating entities based on the green component (“greenValue”) of the color of the current pixel in the level image.

```
private void loadEntities(int greenValue, int x, int y) {
    switch (greenValue) {
        case CRABBY -> crabs.add(new Crabby( x: x * Game.TILES_SIZE,  y: y * Game.TILES_SIZE));
        case PINKSTAR -> pinkstars.add(new Pinkstar( x: x * Game.TILES_SIZE,  y: y * Game.TILES_SIZE));
        case SHARK -> sharks.add(new Shark( x: x * Game.TILES_SIZE,  y: y * Game.TILES_SIZE));
        case 100 -> playerSpawn = new Point( x: x * Game.TILES_SIZE,  y: y * Game.TILES_SIZE);
    }
}
```

The loadEntities method

It is called within the **loadLevel()** method and handles the creations of entities such as “Crabby”, “Pinkstar”, “Shark”, and the player”s spawn point. By doing so, it allows the game to have various entities placed in specific positions within the game level, enabling gameplay interactions and rendering of these entities.

*** loadObjects() method:**

The main purpose of the **loadObjects()** method is to parse the blue component (“blueValue”) of the color of the current pixel in the level image and create the corresponding objects in the game.

```
private void loadObjects(int blueValue, int x, int y) {
    switch (blueValue) {
        case RED_POTION, BLUE_POTION -> potions.add(new Potion( x: x * Game.TILES_SIZE,  y: y * Game.TILES_SIZE, blueValue));
        case BOX, BARREL -> containers.add(new GameContainer( x: x * Game.TILES_SIZE,  y: y * Game.TILES_SIZE, blueValue));
        case SPIKE -> spikes.add(new Spike( x: x * Game.TILES_SIZE,  y: y * Game.TILES_SIZE, SPIKE));
        case CANNON_LEFT, CANNON_RIGHT -> cannons.add(new Cannon( x: x * Game.TILES_SIZE,  y: y * Game.TILES_SIZE, blueValue));
        case TREE_ONE, TREE_TWO, TREE_THREE -> trees.add(new BackgroundTree( x: x * Game.TILES_SIZE,  y: y * Game.TILES_SIZE, blueValue));
    }
}
```

The loadObjects methods

Called within the **loadLevel()** method, it handles the creation of objects such as “Potion”, “GameContainer”, “Spike”, “Cannon” and “BackgroundTree” by the given x and y coordinates. By loading and storing this object data, the game can render and interact with these objects during gameplay.

ii) “LevelManager” class:

The **“LevelManager”** class is responsible for loading, building and management of game levels. It provides the methods that support to load the next level, draw the level on the screen, update the environmental animations and retrieve information about the current level and the total number of levels.

To implement this, we have the LevelManager() constructor to initialize the necessary data by importing sprites (importOutsideSprites() method), creating water animations (createWater() method) and retrieving all the level maps to process (buildAllLevels() method).

```
public LevelManager(Game game) {
    this.game = game;
    importOutsideSprites();
    createWater();
    levels = new ArrayList<>();
    buildAllLevels();
}
```

The LevelManager constructor

Similar to the other management classes, the **“LevelManager”** class consists of three main stages: importing, drawing and updating. The importing stage involves retrieving necessary images by using the “importOutsideSprites()” method, which supports to process the “floor” images, the “buildAllLevels()” method to import and store the map image data and the “createWater()” method to store the animations of water.

```
1 usage  Trần Ngọc Đăng Khôi
private void importOutsideSprites() {
    BufferedImage img = LoadSave.GetSpriteAtlas(LoadSave.LEVEL_ATLAS, Constants.FileType.OBJECT);
    levelSprite = new BufferedImage[48];
    for (int j = 0; j < 4; j++)
        for (int i = 0; i < 12; i++) {
            int index = j * 12 + i;
            levelSprite[index] = img.getSubimage( x: i * 32,  y: j * 32,  w: 32,  h: 32);
        }
}
```

The importOutsideSprites method

```
private void buildAllLevels() {
    BufferedImage[] allLevels = LoadSave.GetAllLevels();
    for (BufferedImage img : allLevels)
        levels.add(new Level(img));
}
```

The buildAllLevels method

```
1 usage  Trần Ngọc Đăng Khôi
private void createWater() {
    waterSprite = new BufferedImage[5];
    BufferedImage img = LoadSave.GetSpriteAtlas(LoadSave.WATER_TOP, Constants.FileType.OBJECT);
    for (int i = 0; i < 4; i++)
        waterSprite[i] = img.getSubimage( x: i * 32,  y: 0,  w: 32,  h: 32);
    waterSprite[4] = LoadSave.GetSpriteAtlas(LoadSave.WATER_BOTTOM, Constants.FileType.OBJECT);
}
```

The createWater method

Moreover, there is also a **draw()** method supports for rendering the level on the screen. It takes a Graphics objects as a parameter, which allows it to draw the level elements using various graphical operations. Based on the sprites index, which retrieved for each tile from the “Level” objects by using the “getSpriteIndex(i,j)” method, to separate the different images:

In case 1: If the sprite index is 48, it indicates a water tile that requires animation. The method selects the appropriate water sprite from the “waterSprite” array based on the “aniIndex” variable and draws it at the calculated position.

In case 2: If the sprite index is 49, it represents the bottom part of the water tile, which remains static. The method draws the corresponding water sprite from the “waterSprite” array.

For any other sprite index, the method retrieves the corresponding image from the “levelSprite” array based on the index and draws it on the screen.

```
public void draw(Graphics g, int lvlOffset) {
    for (int j = 0; j < Game.TILES_IN_HEIGHT; j++)
        for (int i = 0; i < levels.get(lvlIndex).getLevelData()[0].length; i++) {
            int index = levels.get(lvlIndex).getSpriteIndex(i, j);
            int x = Game.TILES_SIZE * i - lvlOffset;
            int y = Game.TILES_SIZE * j;
            if (index == 48)
                g.drawImage(waterSprite[aniIndex], x, y, Game.TILES_SIZE, Game.TILES_SIZE,  observer: null);
            else if (index == 49)
                g.drawImage(waterSprite[4], x, y, Game.TILES_SIZE, Game.TILES_SIZE,  observer: null);
            else
                g.drawImage(levelSprite[index], x, y, Game.TILES_SIZE, Game.TILES_SIZE,  observer: null);
        }
}
```

The draw method

After completed the process and drawing images, we started updating the animations. However, there is only “Water” elements that contains animations, so that, we call directly updateWaterAnimation() method in **update()** method.

```
public void update() { updateWaterAnimation(); }
```

The update method

```
private void updateWaterAnimation() {
    aniTick++;
    if (aniTick >= 40) {
        aniTick = 0;
        aniIndex++;

        if (aniIndex >= 4)
            aniIndex = 0;
    }
}
```

The updateWaterAnimation method

On the other hand, to qualify the “encapsulation” property of OOP, this class also supplies the accessors to provide the corresponding data.

```
11 usages   Trần Ngọc Đăng Khôi
public Level getCurrentLevel() { return levels.get(lvlIndex); }

1 usage   Trần Ngọc Đăng Khôi
public int getAmountOfLevels() { return levels.size(); }

6 usages   HoangHaITITIU21127
public int getLevelIndex() { return lvlIndex; }

2 usages   Trần Ngọc Đăng Khôi
public void setLevelIndex(int lvlIndex) { this.lvlIndex = lvlIndex; }
```

The getter method

c. Entities package:

The purpose of **the "entities" Java package** is to provide a framework for modeling and managing different types of entities in a game with the support of some classes:

i) Entity class:

The **"Entity" abstract class** serves as a blueprint for all entities within the system. It defines the common attributes and behaviors that all entities share. This includes properties like position, hitbox, health, and other characteristics relevant to the game or simulation. The abstract class can also include abstract methods or default implementations for common operations that all entities need to perform.

In detail, this class incorporates several important attributes. It includes variables to represent the entity's position ("x" and "y"), dimensions ("width" and "height"), hitbox ("hitbox"), animation state ("aniTick" and "aniIndex"), current state ("state"), air speed ("airSpeed"), in-air status ("inAir"), maximum health ("maxHealth"), current health ("currentHealth"), attack box ("attackBox"), and walk speed ("walkSpeed").

```
protected float x, y;
protected int width, height;
protected Rectangle2D.Float hitbox;
12 usages
protected int aniTick, aniIndex;
protected int state;
20 usages
protected float airSpeed;
24 usages
protected boolean inAir = false;
9 usages
protected int maxHealth;
13 usages
protected int currentHealth;
18 usages
protected Rectangle2D.Float attackBox;
14 usages
protected float walkSpeed;
```

The attributes of Entity Class

To initialize the entities, we takes parameters to initialize in **“Entity” constructor** by given position (“x” and “y” coordinates) and dimensions (“width” and “height”). These values are provided during instantiation to set the initial properties of the entity. The constructor ensures that the entity is created with the specified position and size, serving as a starting point for defining its behavior and attributes.

```
public Entity(float x, float y, int width, int height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
}
```

The Entity constructor

In some cases that entity is hit by an attack, the **“updatePushBackDrawOffset()” method** determines the amount by which the entity is pushed back. It adjusts the “pushDrawOffset” value based on the “pushBackOffsetDir”, allowing the entity to smoothly return to its original position after being hit.

```
protected void updatePushBackDrawOffset() {
    float speed = 0.95f;
    float limit = -30f;

    if (pushBackOffsetDir == UP) {
        pushDrawOffset -= speed;
        if (pushDrawOffset <= limit)
            pushBackOffsetDir = DOWN;
    } else {
        pushDrawOffset += speed;
        if (pushDrawOffset >= 0)
            pushDrawOffset = 0;
    }
}
```

The updatePushBackDrawOffset method

Additionally, the **"pushBack()" method** is responsible for moving the entity in the opposite direction of the attack. It calculates the horizontal speed ("xSpeed") based on the "pushBackDir" and "walkSpeed" attributes. The method checks if the entity can move to the new position without colliding with obstacles in the level, utilizing the "CanMoveHere()" method, which used to check the movability of entites.

```
protected void pushBack(int pushBackDir, int[][] lvlData, float speedMulti) {
    float xSpeed = 0;
    if (pushBackDir == LEFT)
        xSpeed = -walkSpeed;
    else
        xSpeed = walkSpeed;

    if (CanMoveHere(hitbox.x + xSpeed * speedMulti, hitbox.y, hitbox.width, hitbox.height, lvlData))
        hitbox.x += xSpeed * speedMulti;
}
```

The pushBack method

The **"initHitbox()" method** initializes the hitbox of the entity based on the provided width and height values and represents as a position of the entities on the game.

```
protected void initHitbox(int width, int height) {
    hitbox = new Rectangle2D.Float(x, y, (int) (width * Game.SCALE), (int) (height * Game.SCALE));
}
```

The initHitbox method

For visual representation, the "**drawAttackBox()**" and **"drawHitbox()" methods** are used to draw rectangles on the Graphics object. These rectangles

represent the attack and hitbox areas of the entity, respectively, and are displayed with specific colors, which easily support for developers to check the position of the entities and the area that entities inflict damages during game development.

```
//used to check the Attack/Hitbox area.
no usages  Trần Ngọc Đăng Khôi
protected void drawAttackBox(Graphics g, int xLvlOffset) {
    g.setColor(Color.red);
    g.drawRect((int) (attackBox.x - xLvlOffset), (int) attackBox.y, (int) attackBox.width, (int) attackBox.height);
}

no usages  Trần Ngọc Đăng Khôi
protected void drawHitbox(Graphics g, int xLvlOffset) {
    g.setColor(Color.PINK);
    g.drawRect( x: (int) hitbox.x - xLvlOffset, (int) hitbox.y, (int) hitbox.width, (int) hitbox.height);
}
```

The drawAttackBox and drawHitbox method

Finally, besides the addition of the accessor method of the entity's hitbox, state, and animation index, the **"newState()" method** also exists for setting the state of the entity and resetting the animation variables.

```
//getter
 Trần Ngọc Đăng Khôi
public Rectangle2D.Float getHitbox() { return hitbox; }

 Trần Ngọc Đăng Khôi
public int getState() { return state; }

 Trần Ngọc Đăng Khôi
public int getAniIndex() { return aniIndex; }

//set the state and reset the animation variable.
13 usages  Trần Ngọc Đăng Khôi
protected void newState(int state) {
    this.state = state;
    aniTick = 0;
    aniIndex = 0;
}
```

The getter and newState method

ii) Player class:

The "Player" class represents Captain Jack, the player's avatar in the game, and extends the "Entity" class. It contains several important attributes and methods to manage Captain Jack's behavior and interactions within the game.

This class includes fields such as “animations” array, movement state and status bar attributes for storing Captain Jack's animations, movement states, and status bar UI information, respectively. Variables like “jumpSpeed” and “fallSpeed” control Captain Jack's jumping and gravity mechanics after colliding with objects. The player's health and power bars on the UI are represented by “healthBarWidth” and “healthBarHeight” variables.

```
private BufferedImage[][] animations;
5 usages
private boolean moving = false, attacking = false;
11 usages
private boolean left, right, jump;
```

The animations, movement state attributes of main characters

```
// Jumping / Gravity
1 usage
private float jumpSpeed = -2.25f * Game.SCALE;
1 usage
private float fallSpeedAfterCollision = 0.5f * Game.SCALE;
```

The Jumping and Gravity attributes

```
private BufferedImage statusBarImg;

1 usage
private int statusBarWidth = (int) (192 * Game.SCALE);
1 usage
private int statusBarHeight = (int) (58 * Game.SCALE);
3 usages
private int statusBarX = (int) (10 * Game.SCALE);
3 usages
private int statusBarY = (int) (10 * Game.SCALE);
```

The stats related to Status bar, which includes Health bar and Power bar

```
private int healthBarWidth = (int) (150 * Game.SCALE);
1 usage
private int healthBarHeight = (int) (4 * Game.SCALE);
1 usage
private int healthBarXStart = (int) (34 * Game.SCALE);
1 usage
private int healthBarYStart = (int) (14 * Game.SCALE);
2 usages
private int healthWidth = healthBarWidth;
```

The stats of Health bar

```
private int powerBarWidth = (int) (104 * Game.SCALE);
1 usage
private int powerBarHeight = (int) (2 * Game.SCALE);
1 usage
private int powerBarXStart = (int) (44 * Game.SCALE);
1 usage
private int powerBarYStart = (int) (34 * Game.SCALE);
2 usages
private int powerWidth = powerBarWidth;
4 usages
private int powerMaxValue = 200;
6 usages
private int powerValue = powerMaxValue;
```

The stats of Power bar

To initialize the "Player" object, we create the "**Player" constructor** with player's initial position ("x" and "y" coordinates), "width", "height", and a reference to the "Playing" as the parameters. Within the constructor, various properties of the player are set, such as the player's animations, movement state, current health, and walk speed. After gaining the inputs, we call **the loadAnimation() method** to get and process the animation from the source and initialize the player's hitbox and attack box by **initHitbox()** and **initAttackBox() method**, which support for determining the position of player and area of damage causing.

```
public Player(float x, float y, int width, int height, Playing playing) {
    super(x, y, width, height);
    this.playing = playing;
    this.state = IDLE;
    this.maxHealth = 100;
    this.currentHealth = maxHealth;
    this.walkSpeed = Game.SCALE * 1.0f;
    loadAnimations();
    initHitbox( width: 20,  height: 27);
    initAttackBox();
}
```

The Player constructor

```
private void initAttackBox() {
    attackBox = new Rectangle2D.Float(x, y, (int) (35 * Game.SCALE), (int) (20 * Game.SCALE));
    resetAttackBox();
}
```

The initAttackBox method

```
private void loadAnimations() {
    BufferedImage img = LoadSave.GetSpriteAtlas(LoadSave.PLAYER_ATLAS, FileType.ENTITY);
    animations = new BufferedImage[7][8];
    for (int j = 0; j < animations.length; j++)
        for (int i = 0; i < animations[j].length; i++)
            animations[j][i] = img.getSubimage( x: i * 64,  y: j * 40,  w: 64,  h: 40);

    statusBarImg = LoadSave.GetSpriteAtlas(LoadSave.STATUS_BAR, FileType.OBJECT);
}
```

The loadAnimations method,

Moreover, there are various methods implemented in the “Player” class to handle Captain Jack’s actions and interactions. It includes **update() and render() method**, which update Captain Jack’s position based on player’s input to respond the corresponding actions.

```
public void update() {
    updateHealthBar();
    updatePowerBar();

    if (currentHealth <= 0) {
        if (state != DEAD) {
            state = DEAD;
            aniTick = 0;
            aniIndex = 0;
            playing.setPlayerDying(true);
            playing.getGame().getAudioPlayer().playEffect(AudioPlayer.DIE);

            // Check if player died in air
            if (!IsEntityOnFloor(hitbox, lvlData)) {
                inAir = true;
                airSpeed = 0;
            }
        } else if (aniIndex == GetSpriteAmount(DEAD) - 1 && aniTick >= ANI_SPEED - 1) {
            playing.setGameOver(true);
            playing.getGame().getAudioPlayer().stopSong();
            playing.getGame().getAudioPlayer().playEffect(AudioPlayer.GAMEOVER);
        } else {
            updateAnimationTick();
```

```
            // Check if player died in air
            if (!IsEntityOnFloor(hitbox, lvlData)) {
                inAir = true;
                airSpeed = 0;
            }
        } else if (aniIndex == GetSpriteAmount(DEAD) - 1 && aniTick >= ANI_SPEED - 1) {
            playing.setGameOver(true);
            playing.getGame().getAudioPlayer().stopSong();
            playing.getGame().getAudioPlayer().playEffect(AudioPlayer.GAMEOVER);
        } else {
            updateAnimationTick();

            // Fall if in air
            if (inAir)
                if (CanMoveHere(hitbox.x,  y: hitbox.y + airSpeed, hitbox.width, hitbox.height, lvlData)) {
                    hitbox.y += airSpeed;
                    airSpeed += GRAVITY;
                } else
                    inAir = false;

        }
        return;
    }
```

```
        updateAttackBox();

        if (state == HIT) {
            if (aniIndex <= GetSpriteAmount(state) - 3)
                pushBack(pushBackDir, lvlData, speedMulti: 1.25f);
            updatePushBackDrawOffset();
        } else
            updatePos();

        if (moving) {
            checkPotionTouched();
            checkSpikesTouched();
            checkInsideWater();
            tileY = (int) (hitbox.y / Game.TILES_SIZE);
            if (powerAttackActive) {
                powerAttackTick++;
                if (powerAttackTick >= 35) {
                    powerAttackTick = 0;
                    powerAttackActive = false;
                }
            }
        }

        if (attacking || powerAttackActive)
            checkAttack();

        updateAnimationTick();
        setAnimation();
    }
```

The update method

```
1 usage   Trần Ngọc Đăng Khôi *
public void render(Graphics g, int lvlOffset) {
    g.drawImage(animations[state][aniIndex],
            x: (int) (hitbox.x - xDrawOffset) - lvlOffset + flipX, (int) (hitbox.y - yDrawOffset + (int) (pushDrawOffset)),
            width: width * flipW, height,  observer: null);
//  drawHitbox(g, lvlOffset);
//  drawAttackBox(g, lvlOffset);
    drawUI(g);
}
```

The render method

To handle Captain Jack's actions and interactions more influently, there are some methods that created to support for updating and rendering the animation. For example, to update Captain Jack's position and relevant animation index based on user input, we created and called some methods in “update()” method to support for processing any aspects of the character, such as **updatePos(), updateAnimation(),..** and some methods of updating other character – related elements like attack area, health bar, power bar,….

```
2 usages  Trần Ngọc Đăng Khôi
private void updateAnimationTick() {
    aniTick++;
    if (aniTick >= ANI_SPEED) {
        aniTick = 0;
        aniIndex++;
        if (aniIndex >= GetSpriteAmount(state)) {
            aniIndex = 0;
            attacking = false;
            attackChecked = false;
            if (state == HIT) {
                newState(IDLE);
                airSpeed = 0f;
                if (!IsFloor(hitbox, xSpeed: 0, lvlData))
                    inAir = true;
            }
        }
    }
}
```

The updateAnimationTick method

```
1 usage  Trần Ngọc Đăng Khôi
private void updateAttackBox() {
    if (right && left) {
        if (flipW == 1) {
            setAttackBoxOnRightSide();
        } else {
            setAttackBoxOnLeftSide();
        }

    } else if (right || (powerAttackActive && flipW == 1))
        setAttackBoxOnRightSide();
    else if (left || (powerAttackActive && flipW == -1))
        setAttackBoxOnLeftSide();

    attackBox.y = hitbox.y + (Game.SCALE * 10);
}
```

The updateAttackBox method

```
1 usage  Trần Ngọc Đăng Khôi
private void updateHealthBar() {
    healthWidth = (int) ((currentHealth / (float) maxHealth) * healthBarWidth);
}
```

The updateHealthBar method

```
1 usage  Trần Ngọc Đăng Khôi
private void updatePowerBar() {
    powerWidth = (int) ((powerValue / (float) powerMaxValue) * powerBarWidth);

    powerGrowTick++;
    if (powerGrowTick >= powerGrowSpeed) {
        powerGrowTick = 0;
        changePower( value: 1);
    }
}
```

The updatePowerBar method

There are also some methods that support for checking the interactions of the main character. Like collision checking method, which support for some situations such as **checkPotionTouched(), checkInsideWater() or checkSpikesTouched() methods** to decrease the health of the main character, are also created.

```
1 usage  Trần Ngọc Đăng Khôi
private void checkInsideWater() {
    if (IsEntityInWater(hitbox, playing.getLevelManager().getCurrentLevel().getLevelData()))
        currentHealth = 0;
}

1 usage  Trần Ngọc Đăng Khôi
private void checkSpikesTouched() { playing.checkSpikesTouched( p: this); }

1 usage  Trần Ngọc Đăng Khôi
private void checkPotionTouched() { playing.checkPotionTouched(hitbox); }
```

Some collision checking methods for main character

iii) Enemy class

The "Enemy" class is another specific type of entity that represents non-player characters or opponents in the game. Like the "Player" class, it extends the "Entity" class and inherits its common attributes and behaviors. The "Enemy" class can also have its own unique characteristics and methods, such as algorithms for

checking if player is in attack range, movement mechanics, or any other functionality specific to enemies. There are some important methods within this class:

The "move" method represents the movement logic for enemies in a game. It calculates the movement speed based on the walking direction and checks if the object can move to the new position without collision. If the new position is valid and the floor is valid, the object is moved horizontally. Otherwise, the walking direction is changed, indicating a change in movement behavior.

```
protected void move(int[][] lvlData) {
    float xSpeed = 0;

    if (walkDir == LEFT)
        xSpeed = -walkSpeed;
    else
        xSpeed = walkSpeed;

    if (CanMoveHere(hitbox.x + xSpeed, hitbox.y, hitbox.width, hitbox.height, lvlData))
        if (IsFloor(hitbox, xSpeed, lvlData)) {
            hitbox.x += xSpeed;
            return;
        }

    changeWalkDir();
}
```

The move method

After calling the **"isPlayerInRange" method** to check if the player is within the valid range to be seen. If the player is within range, the method calls the "IsSightClear" method to check if there are no obstacles between the current enemy's hitbox and the player's hitbox at the same vertical tile position ("tileY"). If the sight is clear, meaning no obstacles obstruct the view between the object and the player, the method returns "true", indicating that the player is visible to the enemy.

```
protected boolean canSeePlayer(int[][] lvlData, Player player) {
    int playerTileY = (int) (player.getHitbox().y / Game.TILES_SIZE);
    if (playerTileY == tileY)
        if (isPlayerInRange(player)) {
            if (IsSightClear(lvlData, hitbox, player.hitbox, tileY))
                return true;
        }
    return false;
}
```

The canSeePlayer method

iv) Crabby, Shark, PinkStar class:

The given code snippet contains the implementation of three enemy classes in a game: **Pinkstar**, **Shark**, and **Crabby**. These classes define the behaviors and properties of these enemies in the game by their own **constructor** and **update() method**.

**** Pinkstar class:***

The **Pinkstar class** extends the Enemy class and overrides some of its methods. Since the main type of attack of "**Pinkstar**" enemy is rolling forward to player, it has properties such as "preRoll", "tickSinceLastDmgToPlayer", "tickAfterRollInIdle", "rollDurationTick", and "rollDuration". It has **a constructor** that initializes the Pinkstar enemy with a position and dimensions.

```
public Pinkstar(float x, float y) {
    super(x, y, PINKSTAR_WIDTH, PINKSTAR_HEIGHT, PINKSTAR);
    initHitbox( width: 17,  height: 21);
}
```

The constructor of Pinkstar

The **update() method** is responsible for updating the Pinkstar's behavior and animation. **The updateBehavior() method** handles different states of the Pinkstar, such as IDLE, RUNNING, ATTACK, and HIT. It checks conditions based on the game's level data and player's position to determine the Pinkstar's actions. The Pinkstar can move, attack by rolling, and inflict damage to the player.

```
public void update(int[][] lvlData, Playing playing) {
    updateBehavior(lvlData, playing);
    updateAnimationTick();
}
```

The update method

```
private void updateBehavior(int[][] lvlData, Playing playing) {
    if (firstUpdate)
        firstUpdateCheck(lvlData);

    if (inAir)
        inAirChecks(lvlData, playing);
```

```
    else {
        switch (state) {
            case IDLE:
                preRoll = true;
                if (tickAfterRollInIdle >= 120) {
                    if (IsFloor(hitbox, lvlData))
                        newState(RUNNING);
                    else
                        inAir = true;
                    tickAfterRollInIdle = 0;
                    tickSinceLastDmgToPlayer = 60;
                } else
                    tickAfterRollInIdle++;
                break;
            case RUNNING:
                if (canSeePlayer(lvlData, playing.getPlayer())) {
                    newState(ATTACK);
                    setWalkDir(playing.getPlayer());
                }
                move(lvlData, playing);
                break;
```

```
            case ATTACK:
                if (preRoll) {
                    if (aniIndex >= 3)
                        preRoll = false;
                } else {
                    move(lvlData, playing);
                    checkDmgToPlayer(playing.getPlayer());
                    checkRollOver(playing);
                }
                break;
            case HIT:
                if (aniIndex <= GetSpriteAmount(enemyType, state) - 2)
                    pushBack(pushBackDir, lvlData, speedMulti: 2f);
                updatePushBackDrawOffset();
                tickAfterRollInIdle = 120;

                break;
        }
    }
}
```

The updateBehavior method for Pinkstar

Moreover, to do the rolling of this enemy, we have some functionalities that support for checking, setting and rendering the animations that are tailored to each player's response. Like **checkRollOver() method** is used to check if the Pinkstar enemy has completed a roll attack to set the other state and reset the roll with some emotinal effects for it in **rollOver() method**.

```
private void checkRollOver(Playing playing) {
    rollDurationTick++;
    if (rollDurationTick >= rollDuration) {
        rollOver(playing);
        rollDurationTick = 0;
    }
}
```

The checkRollOver method

```
private void rollOver(Playing playing) {
    newState(IDLE);
    playing.addDialogue((int) hitbox.x, (int) hitbox.y, QUESTION);
}
```

The rollOver method

* ***Shark class:***

The Shark class also extends the Enemy class and overrides some methods. It has a **constructor** that initializes the "**Shark"** enemy with a position and dimensions. The update method updates the behavior, animation, and attack box of the Shark.

```
public Shark(float x, float y) {
    super(x, y, SHARK_WIDTH, SHARK_HEIGHT, SHARK);
    initHitbox( width: 18,  height: 22);
    initAttackBox( w: 20,  h: 20,  attackBoxOffsetX: 20);
}
```

The Shark constructor

The **updateBehavior()** method handles different states of the Shark, including IDLE, RUNNING, ATTACK, and HIT. It checks conditions such as visibility of the player and proximity to trigger specific actions. The Shark can move, attack, and hit the player when in the ATTACK state.

However, there are some differences from the way that Shark attacks the player. Not rolling forward to the main character like “Pinkstar”, firstly, it determines the position of the player if he is in range. Then, Shark enemy starts to use its high speed and jump into the main player. To do that, we create the **attackMove() method** working with the **updateBehavior()** to set the state with the corresponding animations.

```
private void updateBehavior(int[][] lvlData, Playing playing) {
    if (firstUpdate)
        firstUpdateCheck(lvlData);

    if (inAir)
        inAirChecks(lvlData, playing);
    else {
        switch (state) {
            case IDLE:
                if (IsFloor(hitbox, lvlData))
                    newState(RUNNING);
                else
                    inAir = true;
                break;
            case RUNNING:
                if (canSeePlayer(lvlData, playing.getPlayer())) {
                    turnTowardsPlayer(playing.getPlayer());
                    if (isPlayerCloseForAttack(playing.getPlayer()))
                        newState(ATTACK);
                }
```

```
                move(lvlData);
                break;
            case ATTACK:
                if (aniIndex == 0)
                    attackChecked = false;
                else if (aniIndex == 3) {
                    if (!attackChecked)
                        checkPlayerHit(attackBox, playing.getPlayer(), playing);
                    attackMove(lvlData, playing);
                }

                break;
            case HIT:
                if (aniIndex <= GetSpriteAmount(enemyType, state) - 2)
                    pushBack(pushBackDir, lvlData, speedMulti: 2f);
                updatePushBackDrawOffset();
                break;
        }
    }
}
```

The updateBehavior method of Shark

```
protected void attackMove(int[][] lvlData, Playing playing) {
    float xSpeed = 0;

    if (walkDir == LEFT)
        xSpeed = -walkSpeed;
    else
        xSpeed = walkSpeed;

    if (CanMoveHere( x: hitbox.x + xSpeed * 4, hitbox.y, hitbox.width, hitbox.height, lvlData))
        if (IsFloor(hitbox, xSpeed: xSpeed * 4, lvlData)) {
            hitbox.x += xSpeed * 4;
            return;
        }
    newState(IDLE);
    playing.addDialogue((int) hitbox.x, (int) hitbox.y, EXCLAMATION);
}
```

The attackMove method

**** Crabby class:***

The **Crabby class** extends the Enemy class and overrides methods as well. It has a constructor that initializes the Crabby enemy with a position and dimensions.

```
public Crabby(float x, float y) {
    super(x, y, CRABBY_WIDTH, CRABBY_HEIGHT, CRABBY);
    initHitbox( width: 22,  height: 19);
    initAttackBox( w: 82,  h: 19,  attackBoxOffsetX: 30);
}
```

The Crabby constructor

The **update() method** updates the behavior, animation, and attack box of the Crabby. The **updateBehavior() method** handles different states of the "Crabby", including IDLE, RUNNING, ATTACK, and HIT. It checks conditions similar to the other enemy classes to determine the Crabby's behavior. The "Crabby" can move, attack, and hit the player in the ATTACK state.

However, there are some differences from the way that this enemy attacks. Visually, "Crabby" caused damage to player by using his sharp claws to attack horizontally if the player gets close to him. To make this happen, we used some support methods from the **HelpMethod class** of utilz package to check the potions of player and enemy, then active the "ATTACK" mode if player is in the range of enemy and then we loop this process in the **updateBehavior() method**.

```
//update behavior + animations of crabby
Trần Ngọc Đăng Khôi
public void update(int[][] lvlData, Playing playing) {
    updateBehavior(lvlData, playing);
    updateAnimationTick(); //update animation frame
    updateAttackBox();
}
```

The update method

```
//behavior of crabby
1 usage  Trần Ngọc Đăng Khôi
private void updateBehavior(int[][] lvlData, Playing playing) {
    if (firstUpdate)
        firstUpdateCheck(lvlData);

    if (inAir) {
        inAirChecks(lvlData, playing);
    } else {
        switch (state) {
            case IDLE:
                if (IsFloor(hitbox, lvlData))
                    newState(RUNNING);
                else
                    inAir = true;
                break;
            case RUNNING:
                if (canSeePlayer(lvlData, playing.getPlayer())) {
                    turnTowardsPlayer(playing.getPlayer());
                    if (isPlayerCloseForAttack(playing.getPlayer()))
                        newState(ATTACK);
                }
                move(lvlData);
```

```
                if (inAir)
                    playing.addDialogue((int) hitbox.x, (int) hitbox.y, EXCLAMATION);

                break;
            case ATTACK:
                if (aniIndex == 0)
                    attackChecked = false;
                if (aniIndex == 3 && !attackChecked)
                    checkPlayerHit(attackBox, playing.getPlayer(), playing);
                break;
            case HIT:
                if (aniIndex <= GetSpriteAmount(enemyType, state) - 2)
                    pushBack(pushBackDir, lvlData, speedMulti: 2f);
                updatePushBackDrawOffset();
                break;
        }
    }
```

The updateBehavior method of Crabby

v) EnemyManager class:

The "EnemyManager" class has a mission to manage the enemy entities in the game, including their updating and drawing functionalities. It holds references to the different types of enemies in the game, such as crabs, pinkstars, and sharks, and handles their behavior and interactions with the game world.

```
4 usages   Trần Ngọc Đăng Khôi
public class EnemyManager {

    5 usages
    private Playing playing;
    2 usages
    private BufferedImage[][] crabbyArr, pinkstarArr, sharkArr;
    13 usages
    private Level currentLevel;

    //create / update enemy and load them for a particular level
    1 usage   Trần Ngọc Đăng Khôi
    public EnemyManager(Playing playing) {
        this.playing = playing;
        loadEnemyImgs();
    }
```

The "EnemyManager" Class's attributes and constructor

This class also contains methods for loading enemy images, updating the state of enemies based on the level data, and drawing the enemies on the game screen. Therefore, the existence of **the "loadEnemyImgs()" method** in the "EnemyManager" class serves the purpose of loading the enemy images into the game. It retrieves sprite atlases for each enemy type, which are large images containing multiple frames or variations of the enemy sprites arranged in a grid-like structure. The method initializes and populates multidimensional arrays ("crabbyArr", "pinkstarArr", and "sharkArr") to store the individual frames of each enemy type.

```
1 usage   Trần Ngọc Đăng Khôi *
private void loadEnemyImgs() {
    crabbyArr = getImgArr(LoadSave.GetSpriteAtlas(LoadSave.CRABBY_SPRITE, FileType.ENTITY),
            xSize: 9,  ySize: 5, CRABBY_WIDTH_DEFAULT, CRABBY_HEIGHT_DEFAULT);
    pinkstarArr = getImgArr(LoadSave.GetSpriteAtlas(LoadSave.PINKSTAR_ATLAS, FileType.ENTITY),
            xSize: 8,  ySize: 5, PINKSTAR_WIDTH_DEFAULT, PINKSTAR_HEIGHT_DEFAULT);
    sharkArr = getImgArr(LoadSave.GetSpriteAtlas(LoadSave.SHARK_ATLAS, FileType.ENTITY),
            xSize: 8,  ySize: 5, SHARK_WIDTH_DEFAULT, SHARK_HEIGHT_DEFAULT);
}
```

The loadEnemyImgs method

Then, **the "update()" method** iterates through the active enemies and updates their positions and behavior. It also checks if there are any active enemies left in the level and sets the level completion status accordingly.

```
//update the state of enemies
 Trần Ngọc Đăng Khôi
public void update(int[][] lvlData) {
    boolean isAnyActive = false; //used to check any enemies
    for (Crabby c : currentLevel.getCrabs())
        if (c.isActive()) {
            c.update(lvlData, playing);
            isAnyActive = true;
        }

    for (Pinkstar p : currentLevel.getPinkstars())
        if (p.isActive()) {
            p.update(lvlData, playing);
            isAnyActive = true;
        }

    for (Shark s : currentLevel.getSharks())
        if (s.isActive()) {
            s.update(lvlData, playing);
            isAnyActive = true;
        }
    //check levelCompleted
    if (!isAnyActive)
        playing.setLevelCompleted(true);
}
```

The update method

The "draw" method is responsible for rendering the enemies on the screen using the provided "Graphics" object. It loops through the active enemies of each type ("Crabby", "Pinkstars", and "Sharks") and draws their respective images based on their state and animation index.

```
public void draw(Graphics g, int xLvlOffset) {
    drawCrabs(g, xLvlOffset);
    drawPinkstars(g, xLvlOffset);
    drawSharks(g, xLvlOffset);
}
```

The draw method

The "EnemyManager" class also includes a method for checking if any of the enemies have been hit by the player's attack. It takes a rectangle representing the player's attack box and checks for intersections with the hitboxes of the active enemies. If a collision is detected, the corresponding enemy's health is reduced.

```
public void checkEnemyHit(Rectangle2D.Float attackBox) {
    for (Crabby c : currentLevel.getCrabs())
        if (c.isActive())
            if (c.getState() != DEAD && c.getState() != HIT)
                if (attackBox.intersects(c.getHitbox())) {
                    c.hurt( amount: 20);
                    return;
                }
```

The checkEnemyHit method in case of "Crabby" enemy

```
for (Pinkstar p : currentLevel.getPinkstars())
    if (p.isActive()) {
        if (p.getState() == ATTACK && p.getAniIndex() >= 3)
            return;
        else {
            if (p.getState() != DEAD && p.getState() != HIT)
                if (attackBox.intersects(p.getHitbox())) {
                    p.hurt( amount: 20);
                    return;
                }
        }
    }
```

The checkEnemyHit method in case of "Sharks" enemy

```
for (Pinkstar p : currentLevel.getPinkstars())
    if (p.isActive()) {
        if (p.getState() == ATTACK && p.getAniIndex() >= 3)
            return;
        else {
            if (p.getState() != DEAD && p.getState() != HIT)
                if (attackBox.intersects(p.getHitbox())) {
                    p.hurt( amount: 20);
                    return;
                }
        }
    }
```

The checkEnemyHit method in case of "Pinkstar" enemy

Additionally, the "EnemyManager" class has methods for resetting all enemies to their initial state and organizing the image data into a two-dimensional array for efficient access.

```
1 usage   Trần Ngọc Đăng Khôi
public void resetAllEnemies() {
    //resetCrabby
    for (Crabby c : currentLevel.getCrabs())
        c.resetEnemy();

    //resetPinkstar
    for (Pinkstar p : currentLevel.getPinkstars())
        p.resetEnemy();

    //resetShark enemy for new levels
    for (Shark s : currentLevel.getSharks())
        s.resetEnemy();
}
```

The resetAllEnemies method for reseting

```
3 usages   Trần Ngọc Đăng Khôi
private BufferedImage[][] getImgArr(BufferedImage atlas, int xSize, int ySize, int spriteW, int spriteH) {
    BufferedImage[][] tempArr = new BufferedImage[ySize][xSize];
    for (int j = 0; j < tempArr.length; j++)
        for (int i = 0; i < tempArr[j].length; i++)
            tempArr[j][i] = atlas.getSubimage( x: i * spriteW,  y: j * spriteH, spriteW, spriteH);
    return tempArr;
}
```

The getImgArr getter

d. Objects package:

The purpose of **the "Objects" Java package** is to manage and represent objects within a game (containers, potions, trees,...). Objects package include 2 main classes: ObjectManager class and GameObject class.

i) ObjectManager class:

The "ObjectManager" class serves as a central manager for game objects within the system. It is responsible for drawing, and updating every object in the game. The "ObjectManager" class provides methods for adding, removing, and resetting objects images, as well as performing operations that affect multiple objects simultaneously. It may also handle collision detection, object shooting, or other

functionalities related to object management and interaction. Below this are some important methods of ObjectManager class.

This is a method called **"loadImgs()"** that loads images into various arrays for use in the game. By using BufferedImage class and calling the GetSpriteAtlas() method in LoadSave class, we can get the image we want and describe that image with an accessible buffer of image data. As the screenshot below is an example for the algorithm to import and process the image for each objects.

```
private void loadImgs() {
    //potion
    BufferedImage potionSprite = LoadSave.GetSpriteAtlas(LoadSave.POTION_ATLAS, Constants.FileType.OBJECT);
    potionImgs = new BufferedImage[2][7];

    for (int j = 0; j < potionImgs.length; j++)
        for (int i = 0; i < potionImgs[j].length; i++)
            potionImgs[j][i] = potionSprite.getSubimage( x: 12 * i,  y: 16 * j,  w: 12,  h: 16);
```

The loadImgs() method for potion objects

Additionally, this class contains several methods for checking collisions and updating the state of the objects. The **checkSpikesTouched() methods** detect if a player or an enemy has touched a spike object, and apply the appropriate action. The **checkObjectTouched() method** checks if an object has been touched by a hitbox, and if so, applies the effect to the player. Besides that, the **checkObjectHit() method** checks if an object has been hit by an attack box, and triggers an animation and potential creation of a new potion object.

```
public void checkSpikesTouched(Player p) {
    for (Spike s : currentLevel.getSpikes())
        if (s.getHitbox().intersects(p.getHitbox()))
            p.kill();
}

1 usage   Trần Ngọc Đăng Khôi
public void checkSpikesTouched(Enemy e) {
    for (Spike s : currentLevel.getSpikes())
        if (s.getHitbox().intersects(e.getHitbox()))
            e.hurt( amount: 200);
}
```

The checkSpikesTouched method for player and enemy

```
1 usage   Trần Ngọc Đăng Khôi
public void checkObjectTouched(Rectangle2D.Float hitbox) {
    for (Potion p : potions)
        if (p.isActive()) {
            if (hitbox.intersects(p.getHitbox())) {
                p.setActive(false);
                applyEffectToPlayer(p);
            }
        }
}
```

The checkObjectTouched method

```
1 usage   Trần Ngọc Đăng Khôi *
public void checkObjectHit(Rectangle2D.Float attackbox) {
    for (GameContainer gc : containers)
        if (gc.isActive() && !gc.doAnimation) {
            if (gc.getHitbox().intersects(attackbox)) {
                gc.setAnimation(true);
                int type = 0;
                if (gc.getObjType() == BARREL)
                    type = 1;
                potions.add(new Potion((int) (gc.getHitbox().x + gc.getHitbox().width / 2),
                        (int) (gc.getHitbox().y - gc.getHitbox().height / 2), type));
                return;
            }
        }
}
```

The checkObjectHit method

Besides that, this class also handles the loading of object images using sprite atlases and manages the update and drawing of various objects on the game screen. The **update() method** is responsible for updating the state of objects, including potions, containers, cannons, and projectiles. Then, the **draw() method** is created to support for rendering the objects on the screen, including potions, containers, traps, cannons, projectiles, and grass.

```
public void update(int[][] lvlData, Player player) {
    updateBackgroundTrees();
    for (Potion p : potions)
        if (p.isActive())
            p.update();

    for (GameContainer gc : containers)
        if (gc.isActive())
            gc.update();

    updateCannons(lvlData, player);
    updateProjectiles(lvlData, player);
}
```

The update method

```
public void draw(Graphics g, int xLvlOffset) {
    drawPotions(g, xLvlOffset);
    drawContainers(g, xLvlOffset);
    drawTraps(g, xLvlOffset);
    drawCannons(g, xLvlOffset);
    drawProjectiles(g, xLvlOffset);
    drawGrass(g, xLvlOffset);
}
```

The draw method

The final supportive method is the **"resetAllObjects()" method**. It is typically called when there is a need to reset game objects to their default state. It iterates over the lists of potions, containers, and cannons, calling their respective "reset()" methods. This allows the game objects to be reset and prepared for a fresh start or a new level.

```
public void resetAllObjects() {
    loadObjects(playing.getLevelManager().getCurrentLevel());
    for (Potion p : potions)
        p.reset();
    for (GameContainer gc : containers)
        gc.reset();
    for (Cannon c : currentLevel.getCannons())
        c.reset();
}
```

The resetAllObjects method

ii) GameObject class

The **"GameObject" class** provides a basic structure and functionality for game objects, including their position, hitbox, animation, and active state. It also includes methods for updating the animation, resetting the object, initializing the hitbox, and drawing the hitbox. The class can be extended to create specific game objects with additional functionality and behaviors.

The **"updateAnimationTick()" method** is typically called repeatedly within a game loop to update the animation state of an object. It increments the animation tick, checks if it has reached the desired speed, advances the animation index and performs specific actions based on the animation progress and object type.

```
protected void updateAnimationTick() {
    aniTick++;
    if (aniTick >= ANI_SPEED) {
        aniTick = 0;
        aniIndex++;
        if (aniIndex >= GetSpriteAmount(objType)) {
            aniIndex = 0;
            if (objType == BARREL || objType == BOX) {
                doAnimation = false;
                active = false;
            } else if (objType == CANNON_LEFT || objType == CANNON_RIGHT)
                doAnimation = false;
        }
    }
}
```

The updateAnimationTick() method

To visually represent a hitbox by drawing a rectangular outline on the screen, we create the "drawHitbox" method. The hitbox is defined by the "hitbox" object, which likely contains properties such as the x and y coordinates, width, and height of the hitbox. By calling this method and passing the necessary parameters, the hitbox will be drawn on the screen with a pink color outline. This can be useful for debugging or visualizing collision detection in a game.

```
public void drawHitbox(Graphics g, int xLvlOffset) {
    g.setColor(Color.PINK);
    g.drawRect((int) hitbox.x - xLvlOffset, (int) hitbox.y, (int) hitbox.width, (int) hitbox.height);
}
```

The drawHitbox method

Besides the mentioned main classes, there are also some classes which are created to initialize and store the attributes and methods that correspond to the properties and effects of its object, such as "GameContainer" class for the containers

and barrels, Potion class for both health and power bottle, “Cannon” and “Projectiles” class for creating cannon and firing projectiles, “Spike” and “BackgroundTree” class for the background.

e. Gamestates package:

The main purpose of the **Gamestate package** is to store the state of the levels and transfer between them.

i) Gamestate enum:

The given enum named **“Gamestate”** consists of four constants: PLAYING, MENU, OPTIONS, and QUIT. These constants represent different states of a game. It also contains a static variable named “state” of type Gamestate, initialized to MENU. This variable allows for storing and accessing the current state of the game. By using this enum, developers can easily manage and switch between various states of the game, such as when the game is actively being played (PLAYING), in the main menu (MENU), in the options menu (OPTIONS), or when the player chooses to quit (QUIT).

```
package gamestates;

Trần Ngọc Đăng Khôi
public enum Gamestate {
    13 usages
    PLAYING, MENU, OPTIONS, QUIT;

    public static Gamestate state = MENU;
}
```

The Gamestate enum

ii) Statemethods interface:

This interface, which named **“Statemethods”,** aims to encapsulate all the methods required to handle input events related to the game state. It contains a set of methods that cover various types of input events such as mouse and key events. The **“update()” method** is responsible for updating the game state, while the

“draw(Graphics g)” method handles the rendering of the game state. Addititonally, the interface also includes methods, such as “mouseClicked(MouseEvent e)”, “mousePressed(MouseEvent e)”, “mouseReleased(MouseEvent e)”,… for handling mouse event and key events, like “keyPressed(KeyEvent e)” and “keyReleased(KeyEvent e).”

```
public interface Statemethods {
    3 implementations  Trần Ngọc Đăng Khôi
    public void update();

    3 implementations  Trần Ngọc Đăng Khôi
    public void draw(Graphics g);
```

The update and draw method

```
1 usage  3 implementations  Trần Ngọc Đăng Khôi
public void mouseClicked(MouseEvent e);

3 implementations  Trần Ngọc Đăng Khôi
public void mousePressed(MouseEvent e);

3 implementations  Trần Ngọc Đăng Khôi
public void mouseReleased(MouseEvent e);

3 implementations  Trần Ngọc Đăng Khôi
public void mouseMoved(MouseEvent e);

3 usages  3 implementations  Trần Ngọc Đăng Khôi
public void keyPressed(KeyEvent e);

2 usages  3 implementations  Trần Ngọc Đăng Khôi
public void keyReleased(KeyEvent e);
```

The mouse and keyboard action methods

iii) State class:

The given class named "**State**" serves as a base class for all game states within the game. It contains methods and functionality that are common and relevant to game states. The class is part of the "gamestates" package and has a dependency on the "Game" class. The constructor takes an instance of the Game class as a parameter, allowing it to access and interact with the overall game's functionality.

```
public State(Game game) {
    this.game = game;
}
```

The constructor of State

One of the methods in this class is "**isIn()"** , which takes a MouseEvent object and a MenuButton object as parameters. It checks whether the mouse coordinates specified by the MouseEvent are within the bounds of the MenuButton. This method provides a convenient way to determine if a mouse event occurred within a particular button or UI element.

```
3 usages  Trần Ngọc Đăng Khôi
public boolean isIn(MouseEvent e, MenuButton mb) {
    return mb.getBounds().contains(e.getX(), e.getY());
}
```

The isIn method

Another important method in this class is "**setGamestate**," which takes a Gamestate object as a parameter. This method is responsible for changing the game state by updating the static "Gamestate.state" variable. Depending on the specified state, different actions are performed. For example, if the state is MENU, a song is played using **the AudioPlayer class** from the Game object. If the state is PLAYING, the appropriate level song is set based on the current level index. This method allows for dynamic switching between different game states and handles the necessary actions associated with each state.

```
HoangHaiTITIU21127 +1
public void setGamestate(Gamestate state) {
    //change the gamestate
    switch (state) {
        case MENU -> game.getAudioPlayer().playSong(AudioPlayer.MENU_1);
        case PLAYING -> game.getAudioPlayer().setLevelSong(game.getPlaying().getLevelManager().getLevelIndex());
    }
    Gamestate.state = state;
}
```

The setGamestate method

Finally, "**getGame()"** method simply returns the instance of the Game class associated with this state, providing access to the overall game's functionality.

```
public Game getGame() {
    return game;
}
```

The getGame method

iv) Playing class:

The "Playing" class acts as the central manager for the game state while the player is actively playing. It handles player input, updates game elements, and renders them on the screen. It also manages different game states and provides methods to handle game events and transitions between states - The class extends the "**State**" class and implements the "**Statemethods**" interface, indicating its role in managing the game state.

The "Playing" constructor initializes the required classes and loads necessary game assets, including images for the background, clouds, ship, and dialogues.

```
public Playing(Game game) {
    super(game);
    initClasses();

    backgroundImg = LoadSave.GetSpriteAtlas(LoadSave.PLAYING_BG_IMG, Constants.FileType.OBJECT);
    bigCloud = LoadSave.GetSpriteAtlas(LoadSave.BIG_CLOUDS, Constants.FileType.OBJECT);
    smallCloud = LoadSave.GetSpriteAtlas(LoadSave.SMALL_CLOUDS, Constants.FileType.OBJECT);
    smallCloudsPos = new int[8];
    for (int i = 0; i < smallCloudsPos.length; i++)
        smallCloudsPos[i] = (int) (90 * Game.SCALE) + rnd.nextInt((int) (100 * Game.SCALE));

    shipImgs = new BufferedImage[4];
    BufferedImage temp = LoadSave.GetSpriteAtlas(LoadSave.SHIP, Constants.FileType.OBJECT);
    for (int i = 0; i < shipImgs.length; i++)
        shipImgs[i] = temp.getSubimage( x: i * 78,  y: 0,  w: 78,  h: 72);

    loadDialogue();
    calcLvlOffset();
    loadStartLevel();
    setDrawRainBoolean();
}
```

The Playing constructor

This class provides methods to load levels, calculate level offsets, and initialize game classes. However, there is **the "update()" method** which has the essential responsibility for updating the game elements based on the current game state. It updates the overlays, player, level manager, object manager, enemy manager, and checks for the player's proximity to the screen borders. It also updates the ship animation if enabled.

```
Trần Ngọc Đăng Khôi +1
@Override
public void update() {
    if (paused)
        pauseOverlay.update();
    else if (lvlCompleted)
        levelCompletedOverlay.update();
    else if (gameCompleted)
        gameCompletedOverlay.update();
    else if (gameOver)
        gameOverOverlay.update();
    else if (playerDying)
        player.update();
    else {
        updateDialogue();
        if (drawRain)
            rain.update(xLvlOffset);
        levelManager.update();
        objectManager.update(levelManager.getCurrentLevel().getLevelData(), player);
        player.update();
        enemyManager.update(levelManager.getCurrentLevel().getLevelData());
        checkCloseToBorder();
        if (drawShip)
            updateShipAni();
    }
}
```

The update method

Moreover, the **"draw()" method** is also implemented from the "Statemethods" interface and used to render the game elements on the screen. It draws the background, clouds, rain, ship, level elements, objects, enemies, player, dialogue effects, and overlays.

```
@Override
public void draw(Graphics g) {
    g.drawImage(backgroundImg, x: 0, y: 0, Game.GAME_WIDTH, Game.GAME_HEIGHT, observer: null);

    drawClouds(g);
    if (drawRain)
        rain.draw(g, xLvlOffset);

    if (drawShip)
        g.drawImage(shipImgs[shipAni], x: (int) (100 * Game.SCALE) - xLvlOffset,
                (int) ((288 * Game.SCALE) + shipHeightDelta), (int) (78 * Game.SCALE), (int) (72 * Game.SCALE), observer: null);
```

Part of draw method that supports to draw the background, rain, ships

```
levelManager.draw(g, xLvlOffset);
objectManager.draw(g, xLvlOffset);
enemyManager.draw(g, xLvlOffset);
player.render(g, xLvlOffset);
objectManager.drawBackgroundTrees(g, xLvlOffset);
drawDialogue(g, xLvlOffset);
```

The part that support to draw level elements, objects, entities and dialogue

```
    if (paused) {
        g.setColor(new Color( r: 0,  g: 0,  b: 0,  a: 150));
        g.fillRect( x: 0,  y: 0, Game.GAME_WIDTH, Game.GAME_HEIGHT);
        pauseOverlay.draw(g);
    } else if (gameOver)
        gameOverOverlay.draw(g);
    else if (lvlCompleted)
        levelCompletedOverlay.draw(g);
    else if (gameCompleted)
        gameCompletedOverlay.draw(g);

}
```

The part that check to set the corresponding overlay.

The class provides methods to handle mouse clicks, key presses, and mouse movements. It delegates these events to the appropriate overlays based on the current game state. Moreover, the various helper methods are included to manage dialogue effects, clouds, rain, level completion, resetting game state, and checking collisions.

```
1 usage  Trần Ngọc Đăng Khôi
public void checkObjectHit(Rectangle2D.Float attackBox) { objectManager.checkObjectHit(attackBox); }

1 usage  Trần Ngọc Đăng Khôi
public void checkEnemyHit(Rectangle2D.Float attackBox) { enemyManager.checkEnemyHit(attackBox); }

1 usage  Trần Ngọc Đăng Khôi
public void checkPotionTouched(Rectangle2D.Float hitbox) { objectManager.checkObjectTouched(hitbox); }

1 usage  Trần Ngọc Đăng Khôi
public void checkSpikesTouched(Player p) { objectManager.checkSpikesTouched(p); }
```

The collisions checking method

v) **Menu class:**

The provided class is named "**Menu**" and represents the menu state of the game. Its purpose is to display the main menu of the game and handle user inputs related to mouse clicks and mouse movements.

This class is extended from **the "State" class** and implements **the "Statemethods" interface**, indicating its role as a game state and its ability to handle input events. It contains several instance variables, including an array of "MenuButton" objects, two BufferedImage objects for the background images, and variables to store the menu dimensions and position.

```
public class Menu extends State implements Statemethods {

    10 usages
    private MenuButton[] buttons = new MenuButton[3];
    4 usages
    private BufferedImage backgroundImg, backgroundImgPink;
    2 usages
    private int menuX, menuY, menuWidth, menuHeight;
```

The Menu class with its attributes

The constructor initializes the menu buttons, loads the background images, and assigns the pink background image from a sprite atlas. After that, the

"loadBackground()" method is called to calculates the dimensions and position of the menu based on the background image. On the other hand, the **"loadButtons()" method** creates three "MenuButton" objects and sets their positions and associated game states.

The **"update()"**and the **"draw()" method** updates the state and handles the rendering of the menu by drawing the background images and the menu buttons on the graphics object passed as a parameter.

```
Trần Ngọc Đăng Khôi
@Override
public void update() {
    for (MenuButton mb : buttons)
        mb.update();
}

Trần Ngọc Đăng Khôi
@Override
public void draw(Graphics g) {
    g.drawImage(backgroundImgPink, x: 0, y: 0, Game.GAME_WIDTH, Game.GAME_HEIGHT, observer: null);
    g.drawImage(backgroundImg, menuX, menuY, menuWidth, menuHeight, observer: null);

    for (MenuButton mb : buttons)
        mb.draw(g);
}
```

The update and draw method

Moreover, the class also includes event handling methods. As the functionality of the **"mousePressed(MouseEvent e)" method** is to detect if a mouse button is pressed on any of the menu buttons and sets the **"mousePressed"** flag accordingly.

```
Trần Ngọc Đăng Khôi
@Override
public void mousePressed(MouseEvent e) {
    for (MenuButton mb : buttons) {
        if (isIn(e, mb)) {
            mb.setMousePressed(true);
        }
    }
}
```

The mousePressed method

The **"mouseReleased(MouseEvent e)" method** checks if a mouse button is released on a menu button and applies the associated game state if the button was previously pressed. It also sets the level song if the game state is "PLAYING". The **"resetButtons()" method** resets the "mousePressed" flag for all menu buttons.

```
@Override
public void mouseReleased(MouseEvent e) {
    for (MenuButton mb : buttons) {
        if (isIn(e, mb)) {
            if (mb.isMousePressed())
                mb.applyGamestate();
            if (mb.getState() == Gamestate.PLAYING)
                game.getAudioPlayer().setLevelSong(game.getPlaying().getLevelManager().getLevelIndex());
            break;
        }
    }
    resetButtons();
}
```

The mouseReleased method

```
private void resetButtons() {
    for (MenuButton mb : buttons)
        mb.resetBools();
}
```

The resetButtons method

The **"mouseMoved(MouseEvent e)" method** detects mouse movement and sets the "mouseOver" flag for the appropriate menu button if the mouse is within its bounds.

```
Trần Ngọc Đăng Khôi
@Override
public void mouseMoved(MouseEvent e) {
    for (MenuButton mb : buttons)
        mb.setMouseOver(false);

    for (MenuButton mb : buttons)
        if (isIn(e, mb)) {
            mb.setMouseOver(true);
            break;
        }
}
```

The mouseMoved method

f. Inputs package:

The purpose of this package is to provide classes that handle user input events, specifically keyboard and mouse inputs, in the game. The package contains two classes: **"KeyboardInputs"** and **"MouseInputs."**

i) KeyboardInputs class:

The "KeyboardInputs" class has responsibility for handling keyboard inputs by implementing the *KeyListener* [9] interface. It has a dependency on the "GamePanel" class and a reference to an instance of it is stored as a private member variable. Moreover, **the constructor** takes a "GamePanel" object as a parameter and assigns it to the "gamePanel" member variable.

[9] *KeyListener:* used to change the state of key. It is notified against KeyEvent and this interface is found in java.awt.event package.

```
2 usages   Trần Ngọc Đăng Khôi
public class KeyboardInputs implements KeyListener {

    6 usages
    private GamePanel gamePanel;

    1 usage   Trần Ngọc Đăng Khôi
    public KeyboardInputs(GamePanel gamePanel) { this.gamePanel = gamePanel; }
```

The KeyboardInputs class, its attributes and constructor

After the inheritance, the class overrides the three methods defined by the KeyListener interface: "keyReleased(KeyEvent e)", "keyPressed(KeyEvent e)", and "keyTyped(KeyEvent e)":

In **the "keyReleased()" method**, a switch statement is used to determine the current state of the game (stored in the "Gamestate.state" variable). Depending on the game state, the corresponding keyReleased() method of the appropriate game component is called. If the game state is "MENU", the keyReleased() method of the "Menu" object obtained from the "gamePanel" is invoked. If the game state is "PLAYING", the keyReleased() method of the "Playing" object obtained from the "gamePanel" is invoked.

```
Trần Ngọc Đăng Khôi
@Override
public void keyReleased(KeyEvent e) {
    switch (Gamestate.state) {
        case MENU -> gamePanel.getGame().getMenu().keyReleased(e);
        case PLAYING -> gamePanel.getGame().getPlaying().keyReleased(e);
    }
}
```

The keyReleased method

In **the "keyPressed()" method**, a similar switch statement is used to determine the current game state. Depending on the game state, the corresponding keyPressed() method of the appropriate game component is called. If the game state is "MENU", the keyPressed() method of the "Menu" object is invoked. If the game state is "PLAYING", the keyPressed() method of the "Playing" object is invoked. If the game state is "OPTIONS", the keyPressed() method of the "GameOptions" object obtained from the "gamePanel" is invoked.

```
@Override
public void keyPressed(KeyEvent e) {
    switch (Gamestate.state) {
        case MENU -> gamePanel.getGame().getMenu().keyPressed(e);
        case PLAYING -> gamePanel.getGame().getPlaying().keyPressed(e);
        case OPTIONS -> gamePanel.getGame().getGameOptions().keyPressed(e);
    }
}
```

The keyPressed method

However, **the keyTyped method** is not in used, so we let it blank.

```
@Override
public void keyTyped(KeyEvent e) {
    // Not In Use
}
```

The keyTyped method

ii) MouseInput class:

The class, **"MouseInputs"**, is responsible for handling mouse inputs in the game. It implements the "MouseListener" and "MouseMotionListener" interfaces to capture mouse events and acts as an intermediary between the mouse events and the game components, allowing the game panel to respond accordingly based on the current game state.

The class has **a constructor** that takes a "GamePanel" object as a parameter, allowing it to access the game panel and its components.

```
public MouseInputs(GamePanel gamePanel) {
    this.gamePanel = gamePanel;
}
```

The MouseInputs method

The "mouseDragged" method is called when the mouse is dragged. It determines the current game state and delegates the mouse event to the corresponding component in the game panel, depending on the state.

```
1 Tran Ngọc Đang Khoi
@Override
public void mouseDragged(MouseEvent e) {
    switch (Gamestate.state) {
        case PLAYING -> gamePanel.getGame().getPlaying().mouseDragged(e);
        case OPTIONS -> gamePanel.getGame().getGameOptions().mouseDragged(e);
    }
}
```

The mouseDragged method

The "mouseMoved" method is invoked when the mouse is moved. Similar to "mouseDragged", it identifies the game state and forwards the event to the appropriate component based on the state.

```
@Override
public void mouseMoved(MouseEvent e) {
    switch (Gamestate.state) {
        case MENU -> gamePanel.getGame().getMenu().mouseMoved(e);
        case PLAYING -> gamePanel.getGame().getPlaying().mouseMoved(e);
        case OPTIONS -> gamePanel.getGame().getGameOptions().mouseMoved(e);
    }
}
```

The mouseMoved method

The "mouseClicked" method is triggered when the mouse is clicked. It checks the game state and dispatches the event to the corresponding component in the playing state.

```
public void mouseClicked(MouseEvent e) {
    switch (Gamestate.state) {
        case PLAYING -> gamePanel.getGame().getPlaying().mouseClicked(e);
    }
}
```

The mouseClicked method

The "mousePressed" method is called when a mouse button is pressed down. It determines the game state and directs the event to the relevant component, depending on the state.

```
@Override
public void mousePressed(MouseEvent e) {
    switch (Gamestate.state) {
        case MENU -> gamePanel.getGame().getMenu().mousePressed(e);
        case PLAYING -> gamePanel.getGame().getPlaying().mousePressed(e);
        case OPTIONS -> gamePanel.getGame().getGameOptions().mousePressed(e);
    }
}
```

The mousePressed method

The "mouseReleased" method is invoked when a mouse button is released. Like the previous methods, it identifies the game state and handles the event accordingly by calling the appropriate component.

```
Tran Ngọc Đang Khoi
@Override
public void mouseReleased(MouseEvent e) {
    switch (Gamestate.state) {
        case MENU -> gamePanel.getGame().getMenu().mouseReleased(e);
        case PLAYING -> gamePanel.getGame().getPlaying().mouseReleased(e);
        case OPTIONS -> gamePanel.getGame().getGameOptions().mouseReleased(e);
    }
}
```

The mouseReleased method

g. UI package:

Package "UI" was born with the role of storing classes that are related to User's Inteface tasks. It includes in-game, game-stop, kill-out screens and other audio editing tools. To do that, we split into two main groups of classes:

The first set of functions includes classes created to capture, process, and animate buttons used in functional screens. Buttons are generally created with a constructor that takes data related to the button, like coordinates and dimensions, and then gets the images, handles, updates animations and associated functions. Besides,

those classes will be divided according to the function of each button type, like **MenuButton class** to support creating menu keys, or like **PauseButton class** and classes inherited from PauseButton class like **SoundButton, UrmButton** and **VolumeButton class**.

```
public void draw(Graphics g) {
    g.drawImage(imgs[index], xPos - xOffsetCenter, yPos, B_WIDTH, B_HEIGHT, null);
}
```

```
private void loadImgs() {
    imgs = new BufferedImage[3];
    BufferedImage temp = LoadSave.GetSpriteAtlas(LoadSave.MENU_BUTTONS, FileType.MENU);
    for (int i = 0; i < imgs.length; i++)
        imgs[i] = temp.getSubimage(i * B_WIDTH_DEFAULT, rowIndex * B_HEIGHT_DEFAULT, B_WIDTH_DEFAULT, B_HEIGHT_DEFAULT);
}
```

```
public void update() {
    index = 0;
    if (mouseOver)
        index = 1;
    if (mousePressed)
        index = 2;
}
```

The example of draw, loadImgs and update method of the buttons

Next, they need additional background image processing classes to accommodate the corresponding keys, which including **LevelCompletedOverlay, GameOverOverlay, GameCompletedOverlay and PauseOverlay class**. In addition to the basic methods for drawing and updating animations, they include **createImgs()** and **createButtons()** support methods to create and manipulate background effects and buttons based on the given parameters. In addition, there are methods that check and return responses corresponding to the user's mouse action on the function buttons of the table such as **isIn()** method to check or **MouseMoved(), MousePressed()** and **MouseReleased()** method to give specific feedback.

```
private void createButtons() {
    int menuX = (int) (335 * Game.SCALE);
    int playX = (int) (440 * Game.SCALE);
    int y = (int) (195 * Game.SCALE);
    play = new UrmButton(playX, y, URM_SIZE, URM_SIZE, rowIndex: 0);
    menu = new UrmButton(menuX, y, URM_SIZE, URM_SIZE, rowIndex: 2);

}

1 usage  HoangHaiTITIU21127 +1
private void createImg() {
    img = LoadSave.GetSpriteAtlas(LoadSave.DEATH_SCREEN, FileType.MENU);
    imgW = (int) (img.getWidth() * Game.SCALE);
    imgH = (int) (img.getHeight() * Game.SCALE);
    imgX = Game.GAME_WIDTH / 2 - imgW / 2;
    imgY = (int) (100 * Game.SCALE);

}
```

The example for createButton and createImgs method in GameOverOverlay

```
private boolean isIn(UrmButton b, MouseEvent e) {
    return b.getBounds().contains(e.getX(), e.getY());
}
```

The example for checking method isIn

```
public void mouseMoved(MouseEvent e) {
    play.setMouseOver(false);
    menu.setMouseOver(false);

    if (isIn(menu, e))
        menu.setMouseOver(true);
    else if (isIn(play, e))
        play.setMouseOver(true);
}
```

The example for responding method

h. Audio package:

The "Audio" package has a role as a manager of the audio effects in creating, loading and applying in the seperated situations. To do that, the "**AudioPlayer**" class is created for loading and storing the sound effects for the game level.

```
public class AudioPlayer {

    public static int MENU_1 = 0;
    public static int LEVEL_1 = 1;
    public static int LEVEL_2 = 2;

    public static int DIE = 0;
    public static int JUMP = 1;
    public static int GAMEOVER = 2;
    public static int LVL_COMPLETED = 3;
    public static int ATTACK_ONE = 4;
    public static int ATTACK_TWO = 5;
    public static int ATTACK_THREE = 6;
```

Some constant attributes of AudioPlayer

It contains constant value for sound files. It's gonna be a public static so whenever we call this audio player, we don't need access to a getter. Having three different songs consists of MENU_1, LEVEL_1, LEVEL_2, and seven different effects from DIE with index 0, to ATTACK_THREE with index 6.

Having two arrays, these songs and effects are constant values that present the position in the array where these songs or effects will be found.

Additionally, it also has some private variables in the class. The private "Clip" varible, the "currentsongID" is the current song ID to know what kind of song we have in the clip. The private float for volume with "1f " to make the songs and effects loud enough to hear. After that, the private boolean "songMute" and "effectMute" variables to mute the song or effects. The last one, the Random "rand" variable, this random will be used later for when we have attack sounds , it has three different types of attack and that is just to keep the attack sound a little bit different.

```
    private Clip[] songs, effects;
    private int currentSongId;
    private float volume = 1f;
    private boolean songMute, effectMute;
    private Random rand = new Random();
```

The attributes and stats of AudioPlayer

The **AudioPlayer()** constructor is created to perform the Songs and effects. It calls the **loadSongs()** and **loadEffects() methods**.

```
public AudioPlayer() {
    loadSongs();
    loadEffects();
    playSong(MENU_1);
}
```

The AudioPlayer() constructor

```
private void loadSongs()  {
    String[] names = {"menu","level1","level2"};
    songs = new Clip[names.length];
    for (int i =0; i < songs.length;i++)
        songs[i] = getClip(names[i]);
}
```

The loadSongs() method

In **method loadSongs()**. Using String with all the names of all the Songs consists of “menu", “level 1”, “level 2”. Then create a clip array for songs with the length is the index of the songs. Use a statement for this array. For example, the first one will be a menu which is correct the second one will level one, and the third one is level 2.

```
private void loadEffects() {
    String[] effectNames = {"die","jump","gameover","lvlcompleted","attack1","attack2","attack3"};
    effects = new Clip[effectNames.length];
    for (int i =0; i < effects.length;i++)
        effects[i] = getClip(effectNames[i]);

    updateEffectsVolume();

}
```

The loadEffect() method

Similarly to the loadSongs() method, **loadSongEffects()** is created to store the audio of the Effects. We have the String array and effects array depending on the size of that array. The String array "effectNames” must match the order in the instance variable to ensure the sounds for actions of the player not false.

```
private Clip getClip(String name) {
    URL url = getClass().getResource("/audio/" + name + ".wav"); //tim trong file audio
    AudioInputStream audio;

    try {
        audio = AudioSystem.getAudioInputStream(url);
        Clip c = AudioSystem.getClip();
        c.open(audio);  // Open clip and put audio in
        return c;
    } catch (UnsupportedAudioFileException | IOException | LineUnavailableException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    return null;
}
```

The getClip method

The method **getClip()** we're going to use when we are loading a song or an effect. This clip requires a string name, thus after retrieving the file's name from our resource audio folder (“res” file), we return the clip for that particular file. obtaining a URL to those audio files and entering "/audio/" to access the audio folder. To store audio, use the variable AudioInputStream[10]. The audio system will produce a clip for us, and perhaps we could say "c" so that we receive our clip from the audio system, which also ensures that the clip can be utilized in the manner we choose.We are opening the clip and inserting the audio when we say “c.open(audio)”.

In this method, we also using try *catch*[11] method to contain a piece of code that may raise an exception.

```
// mute
public void toggleSongMute() {
    this.songMute = !songMute;
    for (Clip c: songs){
        BooleanControl booleanControl = (BooleanControl) c.getControl(BooleanControl.Type.MUTE);//MUT
    booleanControl.setValue(songMute);
    }
}

public void toggleEffectMute(){
    this.effectMute = !effectMute;
    for (Clip c: effects){
        BooleanControl booleanControl = (BooleanControl) c.getControl(BooleanControl.Type.MUTE);//MUT
    booleanControl.setValue(effectMute);
    }
```

The toggleSongMute() and toggleEffectMute() method

[10] *AudioInputStream*: is an input stream with a specified audio format and length, which including methods to support for obtain, write and convert from an external audio file, stream, or URL.

[11] *Try catch method*: The try statement allows you to define a block of code to be tested for errors while it is being executed. The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The **toggleSongMute()** and **toggleEffectMute()** to mute the songs and the effects of the game. Using the booleanControl to the mute which will be either a false or a true. Going through all clips get a control for each clip and that control gets the value whatever so mute now is.

i. Effect package:

Besides the obligatory environmental elements mentioned earlier, the Effects package has been created to incorporate classes that serve the purpose of adding weather effects and character expressions, thereby increasing the vibrancy and interest in the player"s gaming experience. To do that, this package incorporates two classes with clear purposes: "**DialogueEffect**" and "**Rain**" class.

i) DialogueEffect Class:

The **DialogueEffect Class** is designed to represent a visual effect that is displayed during dialogue in a game. It encapsulates the behavior and attributes of a visual effect and allows for the management of animation, activation and resetting of the effect. Additionally, it provides controlled access to its properties through getter method.

The constructor used to initializes a "**DialogueEffect**" object with provided x,y coordinates, and type of objects.

```
public DialogueEffect(int x, int y, int type) {
    this.x = x;
    this.y = y;
    this.type = type;
}
```

The DialogueEffect constructor

Moreover, we also add **update() method** to update the current animation of the effect. It processes by increments the "aniIndex" of animations with the limit amount of sprites after a certain amount of time ("ANI_SPEED"), which counted by "aniTick".

```
public void update() {
    aniTick++;
    if (aniTick >= ANI_SPEED) {
        aniTick = 0;
        aniIndex++;
        if (aniIndex >= GetSpriteAmount(type)) {
            active = false;
            aniIndex = 0;
        }
    }
}
```

The update method

On the other hand, there are some methods class, like reset(), deactive() or getter method, to support for resetting or deactive the effects in some cases and extracting date intermediately to ensure encapsulation.

```
// turn off
1 usage  Trần Ngọc Đăng Khôi
public void deactive() { active = false; }

// turn on effects and get its intial state with new coordinates
1 usage  Trần Ngọc Đăng Khôi
public void reset(int x, int y) {
    this.x = x;
    this.y = y;
    active = true;
}
```

The deactive and reset method

```
public int getAniIndex() { return aniIndex; }

Trần Ngọc Đăng Khôi
public int getX() { return x; }

Trần Ngọc Đăng Khôi
public int getY() { return y; }

2 usages  Trần Ngọc Đăng Khôi
public int getType() { return type; }

3 usages  Trần Ngọc Đăng Khôi
public boolean isActive() { return active; }
```

The getter method

This class is used to applied the creation of emotional expression of enemy, which are seperated into two types of expression: exclamation and question. It has clearified previously in "Entities" package and each type of enemies has its own conditions to express. For example, "Crabby" is going to express the "question" emotion when the main character dodges its attack.

ii) Rain Class:

The purpose of this class is to create a realistic visual effect of raindrops falling down the screen in a game or interactive application. It handles the management of raindrop positions, their movement simulation, and resetting when they reach the bottom. Additionally, the class is responsible for rendering the raindrop images on the screen, contributing to an immersive rain effect experience for the players.

To do that, firstly, we have the constructor to initialize and store the raindrops as the *Point2D.Float* [12] objects in the "drops" array from the imported image rain particles "rainParticle". Furthermore, raindrops are generated by utilizing the data obtained from the "getRndPos()" method. This method returns a raindrop object with randomized x and y coordinates within specified limits.

[12] *Point2D.Float: is a class that defines a 2D point specified in float precision*

```
public Rain() {
    rand = new Random();
    drops = new Point2D.Float[1000]; // -> store the positions of raindrops
    rainParticle = LoadSave.GetSpriteAtlas(LoadSave.RAIN_PARTICLE, Constants.FileType.OBJECT);
    initDrops();
}
```

The Rain() constructor

```
private void initDrops() {
    for (int i = 0; i < drops.length; i++)
        drops[i] = getRndPos();
}
```

The initDrops() method

```
private Point2D.Float getRndPos() {
    return new Point2D.Float((int) getNewX( xLvlOffset: 0), rand.nextInt(Game.GAME_HEIGHT));
}
```

The getRndPos() method

Then, **the "update()" method** in the "Rain" class is responsible for updating the positions of raindrops by simulating downward movement on the screen. It iterates through each raindrop in the "drops" array and increments their y-coordinate by the defined "rainSpeed". In case that a raindrop reaches the bottom of the screen, its potions is reset to the top with a new x-corditnate determined by the **"getNewX()" method**, taking into account the provided "xLvlOffset" parameters. Therefore, this method ensures the raindrops continue falling and creates a realistic rain effect in the game.

```
public void update(int xLvlOffset) {
    for (Point2D.Float p : drops) {
        p.y += rainSpeed;
        if (p.y >= Game.GAME_HEIGHT) {
            p.y = -20;
            p.x = getNewX(xLvlOffset);
        }
    }
}
```

The update(int xLvlOffset) method

```
private float getNewX(int xLvlOffset) {
    float value = (-Game.GAME_WIDTH) + rand.nextInt((int) (Game.GAME_WIDTH * 3f)) + xLvlOffset;
    return value;
}
```

The getNewX(int xLvlOffset)

The final method to complete the “raindrops” animation is **draw().** As the name, it has a mission that support to drawing the rain particles with the continuously varying x and y coordinate on the screen.

```
public void draw(Graphics g, int xLvlOffset) {
    for (Point2D.Float p : drops)
        g.drawImage(rainParticle, x: (int) p.getX() - xLvlOffset, (int) p.getY(), width: 3, height: 12, observer: null);
}
```

The draw(Graphics g, int xLvlOffset)

j. Utilz package:

The purpose of the **"utilz" package** is to provide helpful methods and constants that can be utilized across the game and can be used by other pakage. Constants class, ObjectConstants, HelpMethod are the most important class in utilz pakage.

i) Constants class:

The "Constants" class serves the purpose of storing and managing constant values that are used throughout the game. It typically includes values such as screen dimensions, game settings, character states, or any other parameters that are fixed and shared across different components of the game. By centralizing these constants in a dedicated class, it allows for easy modification and consistency throughout the codebase.

```java
public class Constants {

    3 usages
    public static final float GRAVITY = 0.035f * Game.SCALE;
    7 usages
    public static final int ANI_SPEED = 25;

    Trần Ngọc Đăng Khôi
    public static class FileType {
        public static final String MENU = "menu";
        6 usages
        public static final String ENTITY = "entity";
        17 usages
        public static final String OBJECT = "obj";
    }
```

The example for representing the constants-storing functionality

ii) LoadSave class:

The purpose of **the "LoadSave" class** is to provide methods for loading and retrieving various sprites and images used in the game. It contains a list of constants that represent the file names of the sprites and many methods that support for importing the images.

```java
1 usage
public static final String PLAYER_ATLAS = "player_sprites.png";
1 usage
public static final String LEVEL_ATLAS = "outside_sprites.png";
1 usage
public static final String MENU_BUTTONS = "button_atlas.png";
1 usage
public static final String MENU_BACKGROUND = "menu_background.png";
1 usage
public static final String PAUSE_BACKGROUND = "pause_menu.png";
1 usage
public static final String SOUND_BUTTONS = "sound_button.png";
1 usage
public static final String URM_BUTTONS = "urm_buttons.png";
1 usage
public static final String VOLUME_BUTTONS = "volume_buttons.png";
2 usages
public static final String MENU_BACKGROUND_IMG = "background_menu.png";
1 usage
```

```
Trần Ngọc Đăng Khôi
public static BufferedImage GetSpriteAtlas(String fileName, String fileType) {
    BufferedImage img = null;
    InputStream is = LoadSave.class.getResourceAsStream( name: "/" + fileType + "/" + fileName);
    try {
        img = ImageIO.read(is);

    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            is.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    return img;
}
```

The example for the reference-storing and importing functionality

iii) HelpMethod class:

The "HelpMethods" class, with helpful methods, assists in handling movement, collision detection, and visibility aspects within the game environment. By encapsulating these functionalities within a dedicated class, it promotes code reusability, organization, and simplifies the implementation of game logic related to movement, interactions, and line of sight calculations.

The class begins with the **CanMoveHere() method**, which checks if a given position and dimensions (x, y, width, height) are valid for movement based on the level data provided. It iteratively checks if each corner of the entity's bounding box is not solid, indicating that movement is possible.

```
public static boolean CanMoveHere(float x, float y, float width, float height, int[][] lvlData) {
    if (!IsSolid(x, y, lvlData))
        if (!IsSolid( x: x + width,  y: y + height, lvlData))
            if (!IsSolid( x: x + width, y, lvlData))
                if (!IsSolid(x,  y: y + height, lvlData))
                    return true;
    return false;
}
```

The CanMoveHere method

Moreover, there is **the IsSolid() method** is a helper method used by other methods to determine if a specific position is solid based on the level data. It

considers the boundaries of the level and converts the coordinates into indices for the “lvlData” array, checking if the tile at that position is solid.

```
private static boolean IsSolid(float x, float y, int[][] lvlData) {
    int maxWidth = lvlData[0].length * Game.TILES_SIZE;
    if (x < 0 || x >= maxWidth)
        return true;
    if (y < 0 || y >= Game.GAME_HEIGHT)
        return true;
    float xIndex = x / Game.TILES_SIZE;
    float yIndex = y / Game.TILES_SIZE;

    return IsTileSolid((int) xIndex, (int) yIndex, lvlData);
}
```

The IsSolid method

Next, the class provides methods such as **IsProjectileHittingLevel, IsEntityInWater,** and **IsEntityOnFloor** that check specific conditions for projectiles, entities in water, and entities on the floor, respectively. These methods use the **IsSolid() method** to perform the necessary checks based on the provided bounding boxes or hitboxes.

```
public static boolean IsProjectileHittingLevel(Projectile p, int[][] lvlData) {
    return IsSolid( x: p.getHitbox().x + p.getHitbox().width / 2,  y: p.getHitbox().y + p.getHitbox().height / 2, lvlData);
}
```

The IsProjectileHittingLevel method

```
2 usages   Trần Ngọc Đăng Khôi
public static boolean IsEntityInWater(Rectangle2D.Float hitbox, int[][] lvlData) {
    // Will only check if entity touch top water. Can't reach bottom water if not
    // touched top water.
    if (GetTileValue(hitbox.x,  yPos: hitbox.y + hitbox.height, lvlData) != 48)
        if (GetTileValue( xPos: hitbox.x + hitbox.width,  yPos: hitbox.y + hitbox.height, lvlData) != 48)
            return false;
    return true;
}
```

The IsEntityInWater method

```
public static boolean IsEntityOnFloor(Rectangle2D.Float hitbox, int[][] lvlData) {
    if (!IsSolid(hitbox.x, hitbox.y + hitbox.height + 1, lvlData))
        if (!IsSolid(hitbox.x + hitbox.width, hitbox.y + hitbox.height + 1, lvlData))
            return false;
    return true;
}
```

The IsEntityOnFloor method

Additionally, this class also includes methods like **GetEntityXPosNextToWall** and **GetEntityYPosUnderRoofOrAboveFloor** that determine the precise position of an entity next to a wall or under a roof/floor based on its hitbox and movement speed. These methods calculate and return the adjusted position values.

```
1 usage   Trần Ngọc Đăng Khôi
public static float GetEntityXPosNextToWall(Rectangle2D.Float hitbox, float xSpeed) {
    int currentTile = (int) (hitbox.x / Game.TILES_SIZE);
    if (xSpeed > 0) {
        // Right
        int tileXPos = currentTile * Game.TILES_SIZE;
        int xOffset = (int) (Game.TILES_SIZE - hitbox.width);
        return tileXPos + xOffset - 1;
    } else
        // Left
        return currentTile * Game.TILES_SIZE;
}
```

The GetEntityXPosNextToWall method

```
2 usages   Trần Ngọc Đăng Khôi
public static float GetEntityYPosUnderRoofOrAboveFloor(Rectangle2D.Float hitbox, float airSpeed) {
    int currentTile = (int) (hitbox.y / Game.TILES_SIZE);
    if (airSpeed > 0) {
        // Falling - touching floor
        int tileYPos = currentTile * Game.TILES_SIZE;
        int yOffset = (int) (Game.TILES_SIZE - hitbox.height);
        return tileYPos + yOffset - 1;
    } else
        // Jumping
        return currentTile * Game.TILES_SIZE;

}
```

The GetEntityYPosNextToWall method

Furthermore, there are methods like **IsFloor, CanCannonSeePlayer, IsAllTilesClear,** and **IsAllTilesWalkable** that handle more specific checks related to floors, visibility between objects, and tile walkability. These methods consider tile values and ranges to determine the results of the checks.

```
7 usages  Trần Ngọc Đăng Khôi
public static boolean IsFloor(Rectangle2D.Float hitbox, float xSpeed, int[][] lvlData) {
    if (xSpeed > 0)
        return IsSolid( x: hitbox.x + hitbox.width + xSpeed,  y: hitbox.y + hitbox.height + 1, lvlData);
    else
        return IsSolid( x: hitbox.x + xSpeed,  y: hitbox.y + hitbox.height + 1, lvlData);
}
```

The IsFloor method

```
public static boolean CanCannonSeePlayer(int[][] lvlData, Rectangle2D.Float firstHitbox,
                                         Rectangle2D.Float secondHitbox, int yTile) {
    int firstXTile = (int) (firstHitbox.x / Game.TILES_SIZE);
    int secondXTile = (int) (secondHitbox.x / Game.TILES_SIZE);

    if (firstXTile > secondXTile)
        return IsAllTilesClear(secondXTile, firstXTile, yTile, lvlData);
    else
        return IsAllTilesClear(firstXTile, secondXTile, yTile, lvlData);
}
```

The CanCannonSeePlayer method

```
3 usages  Trần Ngọc Đăng Khôi
public static boolean IsAllTilesClear(int xStart, int xEnd, int y, int[][] lvlData) {
    for (int i = 0; i < xEnd - xStart; i++)
        if (IsTileSolid( xTile: xStart + i, y, lvlData))
            return false;
    return true;
}

2 usages  Trần Ngọc Đăng Khôi
public static boolean IsAllTilesWalkable(int xStart, int xEnd, int y, int[][] lvlData) {
    if (IsAllTilesClear(xStart, xEnd, y, lvlData))
        for (int i = 0; i < xEnd - xStart; i++) {
            if (!IsTileSolid( xTile: xStart + i,  yTile: y + 1, lvlData))
                return false;
        }
    return true;
}
```

The IsAllTilesClear and IsAllTilesWalkable method

CHAPTER 4:
CONCLUSION

1. Result:

To summarize, the development of the Treasure Hunt game was made easier and more logical by adopting the Object-Oriented Programming (OOP) approach. This project effectively exemplifies the fundamental principles of OOP, such as encapsulation, abstraction, inheritance, and polymorphism. Alongside gaining expertise in applying OOP principles, it is essential to expand knowledge in related areas beyond the course curriculum. These areas encompass utilizing GIT for version control, resolving software bugs, solving complex problems, programming interactive games, and improving collaboration skills.

2. Limitation:

Furthermore, the game possesses several constraints. To be specific, the game remains unfinished due to time constraints and a lack of sufficient experience and expertise. Consequently, the classes and methods lack proper organization, efficiency, and cleanliness, making testing a rather complex task. Most importantly, the team encountered challenges in constructing, coordinating, and assigning tasks, occasionally necessitating starting over due to inexperience and confusion in the game development process.

3. Performance review table:

No.	Full Name	Coding	Report	Overall
1	Trần Ngọc Đăng Khôi	100%	100%	100%
2	Nguyễn Trần Hoàng Hạ	100%	100%	100%
3	Hà Văn Uyển Nhi	100%	100%	100%
4	Nguyễn Hoàng Quân	100%	100%	100%

REFERENCES

1. libGDX: libGDX
2. LWJGL: LWJGL - Lightweight Java Game Library
3. GeeksforGeeks: Java Swing - JPanel With Examples - GeeksforGeeks
4. Codespeedy.com: What is JPanel in Java with Examples - CodeSpeedy
5. Oracle.com: BufferedImage (Java Platform SE 8) (oracle.com)
6. GeeksforGeeks: What is Java AWT Graphics? - GeeksforGeeks
7. Resource of the game's image: Treasure Hunters by Pixel Frog (itch.io)
8. Java Game Development Tutorial – Kaarin Gaming: https://youtu.be/6_N8QZ47toY
9. WindowFocusListener – oracle.com: WindowFocusListener (Java Platform SE 8) (oracle.com)
10. Uses of InterFace java.awt.event.WindowFocusListener – oracle.com: Uses of Interface java.awt.event.WindowFocusListener (Personal Basis Profile 1.1.2) (oracle.com)
11. RGB and Color Code in Java - Javatpoint: Java Color Codes - Javatpoint
12. Colors in Java: https://teaching.csse.uwa.edu.au/units/CITS1001/colorinfo.html
13. Thread in Java – w3schools: Java Threads (w3schools.com)
14. Runnable interface – upgrad.com: Runnable Interface in Java: Implementation, Steps & Errors upgrad.com https://www.upgrad.com › Blog › Full Stack Development
15. AudioInPutStrean - Oracle.com: AudioInputStream (Java Platform SE 8) (oracle.com)
16. KeyListener – Javatpoint: Java KeyListener - javatpoint
17. Point2D.Float class – oracle.com: Point2D.Float (Java Platform SE 8) (oracle.com)