\

BLUE BOY ADVENTURE PROJECT REPORT

The goal of the game is to find the hidden treasure by traveling to specific locations, collecting items, and returning them to the treasure chest. Players must explore different areas, solve challenges, and gather clues to progress. Along the way, they can pick up tools or artifacts that help them overcome obstacles and reach their destinations. The objective is to return to the treasure chest with all the collected items to unlock the final reward and win the game.

Contents

Contents

CHAP 1: INTRODUCTION

1.1 From being simple to complicate

Video game development typically begins with basic ideas that gradually evolve into complex systems, supported by the features of programming algorithms. Initially, games are created as prototype versions to test fundamental mechanics, such as interactions between objects. As development progresses, additional elements like classes, objects, mechanics, user-friendly interfaces, and advanced principles of Object-Oriented Programming are incorporated to enhance the structure and functionality of the code. Moreover, aspects such as visuals, animations, audio, and user experience bring depth to the project, requiring teamwork across multiple fields. This evolution from initial mechanics to a polished final product highlights the iterative process of game development, where constant improvements turn a concept into a captivating and immersive experience.

By working on a project like ***BLUE BOY ADVENTURE*** for the "Object-Oriented Programming" course, we aim not only to earn full marks but also to strengthen our Java programming expertise.

1.2 About our game project

Our game is inspired by the popular title *BLUE BOY ADVENTURE* originally developed by RyiSnow. Given the constraints of time and our current development skills, we chose to create a simplified version of the game while preserving its core concept. To make the game accessible, we focused on simplifying its mechanics and gameplay, ensuring it is easy to understand and enjoyable while serving as an educational tool. Additionally, this streamlined approach allows for easier enhancements and modifications in the future. In conclusion, although our version cannot match the scale and complexity of the original *BLUE BOY ADVENTURE*, it provides a fun and engaging experience for those looking for a brief moment of relaxation.

1.3 The initial concept of the game

According to point 1.2, we developed a simplified version of Blue Boy Adventure in Java. The objective of the game is to discover hidden treasure by navigating locations, gathering items, and delivering them to the treasure chest. To advance, players guide a character to retrieve a key, collect it, and return it to the treasure chest to proceed to the next level. Along the journey, they can utilize tools and overcome obstacles to achieve their objectives and complete the game.

1.4 Reference

- Image from: https://drive.google.com/drive/folders/1OBRM8M3qCNAfJDCaldg62yFMiyFaKgYx
- The tutorial Java coding:

  https://www.youtube.com/@RyiS
- The resource: https://github.com/maiTienHung/Game_2

1.5 Developer Team

Name	ID	Contribute
Mai Tien Hung	ITITWE22115	Coding, fixes bug
Tran Vo The Vinh	ITITWE22124	Coding, write report

CHAP 2: SOFTWARE REQUIREMENTS

2.1 What we have

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

2.2 Our Objectives

1. Full-Scale High Definition.
2. Construct the Whole System with Optimal Efficiency.
3. Offer an Editable Catalog of Plants and Zombies with Distinct Powers
4. Simple to Modify.

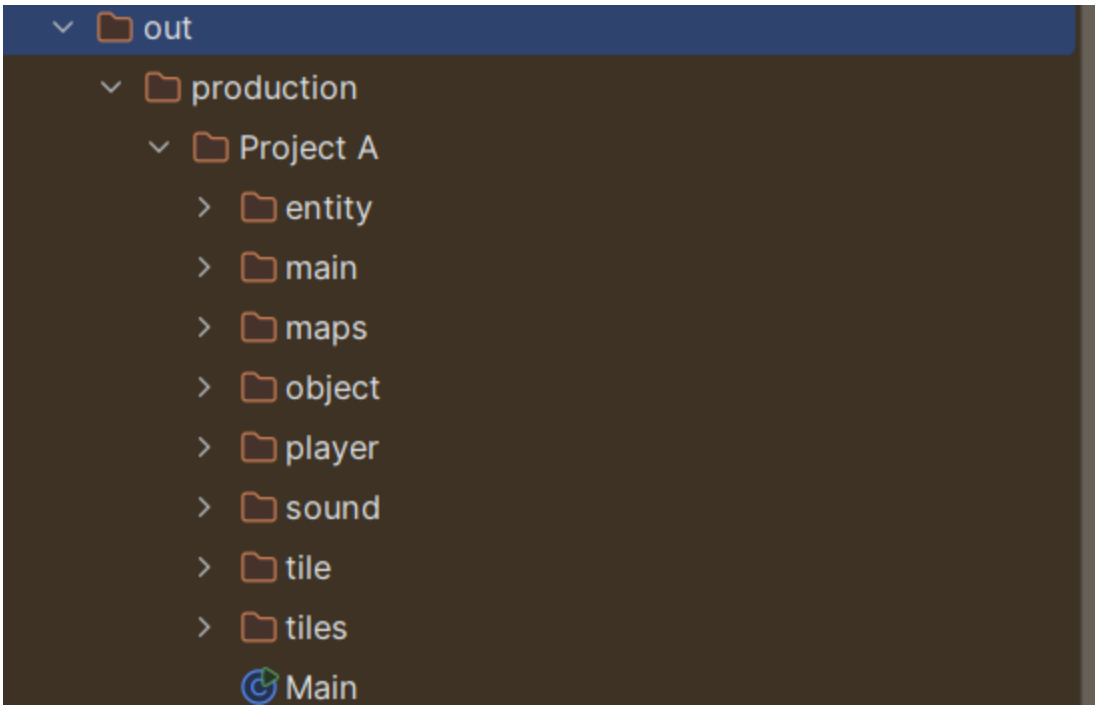
2.3 Working platform

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

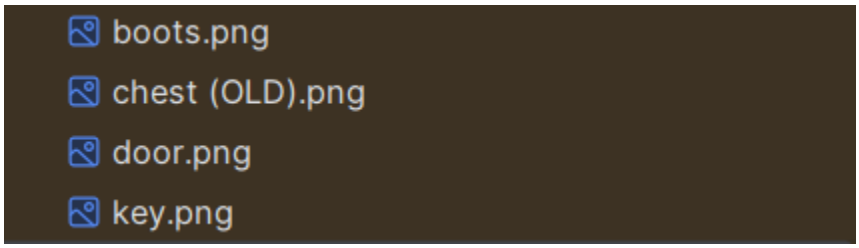
CHAP 3: DESIGN & IMPLEMENTATION

Package Diagram

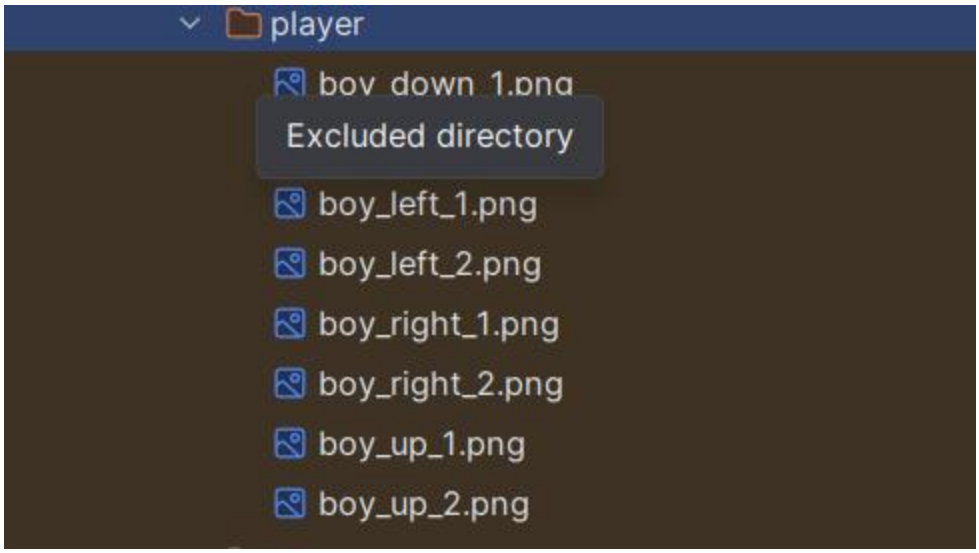
We chose to store our assets in an **out** folder with **res** folders such as map, object, player, sound and tiles to simplify their management. Furthermore, we transfer them to the src folder for use in the program.



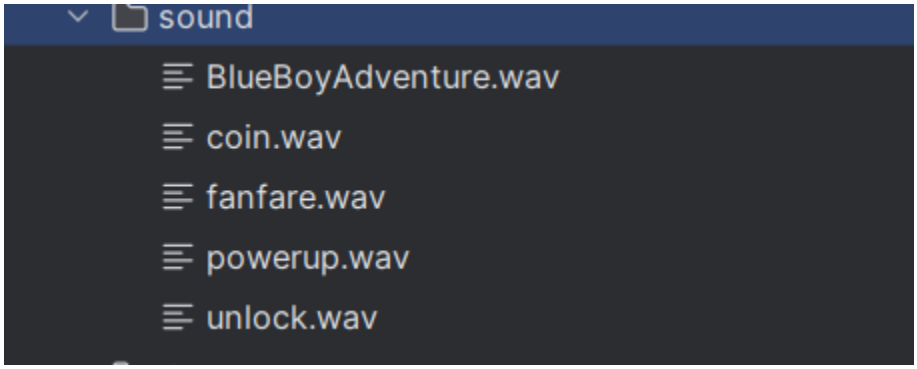
- **Maps:** the folder contains maps.
- **Object:** the folder contains png file of boots, chest, door, key.



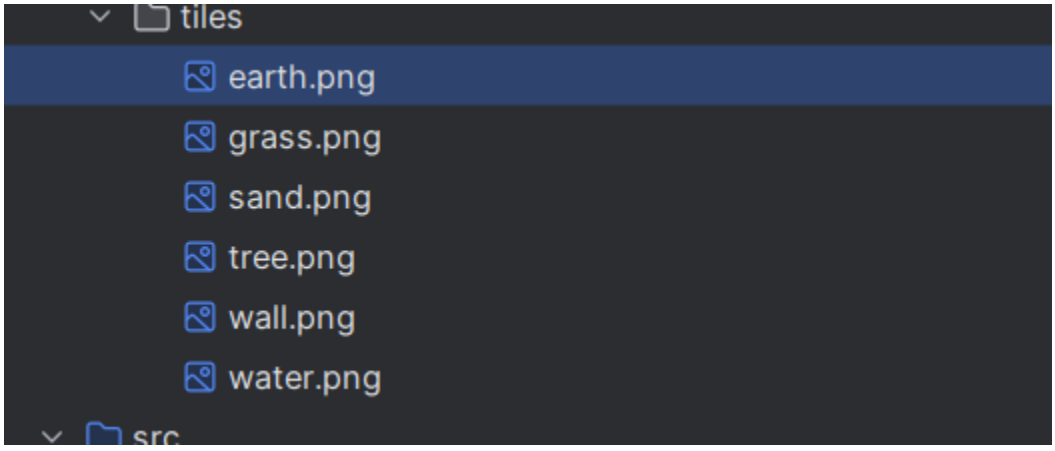
- **Player:** the folder contains png file of main characters.



- **Sound:** the folder contains MP3 file of sound related to game.



- **Tile:** the folder contains png file of every thing on the map.



In terms of UI design, we use the timer, actionevent, and Jpanel from the Java package to efficiently create the UI within the code and pre-configure screen elements.

Java Package

1. Entity:

+ Entity:

Declares the package entity in which the class is located, arranging the code to improve modularity and maintainability.

```
package entity;
```

- **java.awt.*:** offers classes for handling visuals (such as Rectangle) and building graphical user interfaces (GUIs).

- **java.awt.image.BufferedImage**: Used for handling and manipulating images

```
import java.awt.*;
import java.awt.image.BufferedImage;
```

- Defines the Entity public class, which is usable in other program sections. In a game or simulation, it could stand in for a character, item, or unit.

```
public class Entity {
```

- **worldX and worldY**: Represent the position of the entity in the world, likely in a 2D coordinate system.

- **speed**: Represents how fast the entity moves, likely measured in units per tick

```
public int worldX, worldY;  12 usages
public int speed;  14 usages
```

- Stores images for the entity's animations:
    - **up1 and up2**: Images for moving upward.
    - **down1 and down2**: Images for moving downward.
    - **right1 and right2**: Images for moving to the right.
    - **left1 and left2**: Images for moving to the left.
- Having two images for each direction suggests alternating frames for a walking animation.

stores the current direction of the entity as a string, e.g., "up", "down", "left", "right".

```
public BufferedImage up1, up2, down1, down2, right1, right2, left1, lef
public String direction;  9 usages
```

- **spriteCounter**: Tracks time or updates for changing the animation frame, typically incremented in the game loop.

- **spriteNum**: Determines the current animation frame (1 or 2).

```
public int spriteCounter = 0;  3 usages
public int spriteNum = 1;  12 usages
```

 solidArea: The hitbox, or collision boundary, of the entity is defined by a rectangle. For the purpose of detecting collisions.

 solidAreaDefaultX and solidAreaDefaultY: Save the solidArea's default offset from the position of the entity.

 collisionOn: a boolean flag that indicates if a collision is taking place.

```
public Rectangle solidArea;  27 usages
public int solidAreaDefaultX, solidAreaDefaultY;  2 usages
public boolean collsionOn = false;  10 usages
```

+ Player:

```
import main.KeyHandler;

import java.awt.*;
import java.awt.image.BufferedImage;
import java.io.IOException;

import javax.imageio.ImageIO;

import main.GamePanel;
```

- **main.KeyHandler**: manages keyboard input, including determining when the player is moving by pressing the arrow keys.

- **java.awt.***: Provides graphical tools:

- Graphics2D: For rendering graphics.
- Rectangle: Used for defining the player's collision area.

- **java.awt.image.BufferedImage**: BufferedImage objects are used to hold the player's animation frames, such as walking up, down, left, or right.

- **javax.imageio.ImageIO**: Loads image resources (e.g., sprites).

- **main.GamePanel** : Represents the main game area, managing elements like the game loop, objects, and rendering.

```
public class Player extends Entity{  3 usages
```

```
GamePanel gp;  22 usages
KeyHandler keyH;  9 usages
```

-gp: A reference to the GamePanel for accessing game settings and objects.

-**keyH**: Handles keyboard inputs for player movement.

```
public final int screenX;  8 usages
public final int screenY;  8 usages
```

- screenX and screenY: Define the player's fixed position on the screen (centered by dividing screen dimensions and adjusting for tile size).

```
public int hasKey = 0;  4 usages
```

hasKey: Tracks how many keys the player has collected.

```
public Player(GamePanel gp, KeyHandler keyH){  1 usage
```

-**GamePanel gp:**

a pointer to the main game area, represented by the GamePanel object.
- **KeyHandler keyH**:

a reference to the KeyHandler object, which manages keyboard inputs from the player

```
this.gp = gp;
this.keyH = keyH
```

Saves the references to GamePanel and KeyHandler so the Player class can use them later

```
screenX = gp.screenWidth / 2 - (gp.titleSize/2);
screenY = gp.screenHeight / 2 - (gp.titleSize/2);
```

- **Player** is Fixed at the Center of the Screen

  - No matter where the player moves in the game world, they always appear at a fixed point on the screen: the center.
  - The surrounding environment (tiles, objects, enemies) scrolls to give the illusion of movement.

- The Game World Moves Around the Player

  - The player's position in the game world is tracked using world coordinates (worldX and worldY).
  - The screen coordinates (screenX and screenY) are where the player's sprite is drawn, which is always in the middle of the screen.

```
solidArea = new Rectangle();
solidArea.x = 8;
solidArea.y = 16;
solidAreaDefaultX = solidArea.x;
solidAreaDefaultY = solidArea.y;
solidArea.width = 32;
solidArea.height = 32;

setDefaultValues();
getPlayerImage();
```

-solidArea = new Rectangle();

Creates a new Rectangle object to represent the player's collision box

-Position (x and y)

The collision box's top-left corner is offset within the player sprite.
x = 8: The collision box is 8 pixels to the right of the sprite's left border.
y = 16: The collision box begins 16 pixels beneath the sprite's top edge.

-Defaults for Resetting (solidAreaDefaultX and solidAreaDefaultY):

These store the default offsets for the collision box, which can be restored if the solidArea is modified during gameplay (e.g., a power-up might temporarily enlarge the box).

-Size (width and height):

The collision box is 32 pixels wide and 32 pixels tall, which might be smaller than the full sprite (e.g., a 48x48 tile).

-setDefaultValues();

The setDefaultValues method positions the player within the globe (worldX and worldY).
Determines the player's speed and beginning direction.

-getPlayerImage();

Calls the getPlayerImage method to load sprite images for different directions (up, down, left, right).

```
public void setDefaultValues(){  1 usage

    worldX= gp.titleSize * 23;
    worldY= gp.titleSize * 21;
    speed = 4;
    direction = "down";
```

worldX and worldY:

- *Represent the player's coordinates in the larger game world.*
- *Unlike screenX and screenY, which are fixed on the screen, these values track the player's position relative to the entire game map*

speed = 4;

Purpose: Sets how many pixels the player moves per frame.

A value of 4 means the player moves 4 pixels at a time when a movement key is pressed.

direction = "down";

Determines which sprite image is shown when the game starts

```
public void getPlayerImage(){  1 usage
    try {
        up1 = ImageIO.read(getClass().getResourceAsStream( name: "/player/boy_up_1.png"));
        up2 = ImageIO.read(getClass().getResourceAsStream( name: "/player/boy_up_2.png"));
        down1 = ImageIO.read(getClass().getResourceAsStream( name: "/player/boy_down_1.png"));
        down2 = ImageIO.read(getClass().getResourceAsStream( name: "/player/boy_down_2.png"));
        left1 = ImageIO.read(getClass().getResourceAsStream( name: "/player/boy_left_1.png"));
        left2 = ImageIO.read(getClass().getResourceAsStream( name: "/player/boy_left_2.png"));
        right1 = ImageIO.read(getClass().getResourceAsStream( name: "/player/boy_right_1.png"));
        right2 = ImageIO.read(getClass().getResourceAsStream( name: "/player/boy_right_2.png"));
```

*The **getPlayerImage()** method loads the sprite images from the resources folder into memory. These sprites are used to animate the player's movement in multiple directions.*

***getClass().getResourceAsStream()**:*

Finds the file path (/player/boy_up_1.png) relative to the resources folder in the project.

***ImageIO.read()**:*

Reads the image file and converts it into a BufferedImage object that can be used in the game.

Direction	Variable	Direction
Up	up1, up2	/player/boy_up_1.png, /player/boy_up_2.png
Down	down1, down2	/player/boy_down_1.png, /player/boy_down_2.png
Left	left1, left2	/player/boy_left_1.png, /player/boy_left_2.png
Right	right1, right2	/player/boy_right_1.png, /player/boy_right_2.png

```
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Handles any exceptions that might occur if the image files cannot be loaded (e.g., file missing, wrong path).

e.printStackTrace():

- *Prints the error to the console for debugging.*

```
public void update() {  1 usage
```

Throughout the game loop, player movements and actions are managed by the ***update()*** *method. The player's position is updated, collisions are detected, sprite animation is controlled, and user input is analyzed.*

```
if (keyH.upPressed == true) {
    direction = "up";
} else if (keyH.downPressed == true) {
    direction = "down";
} else if (keyH.leftPressed == true) {
    direction = "left";
} else if (keyH.rightPressed == true) {
    direction = "right";
}
```

This block of code checks which movement key the player has pressed and updates the player's direction accordingly. The direction variable is later used to determine both:

1. *Movement: Adjusting the player's position (worldX and worldY).*
2. *Rendering: Choosing the correct sprite image to display.*

How It Works

If the "Up" key is pressed:

- ***`keyH.upPressed == true`*** *evaluates as* `true`.
- *The* `direction` *is set to* `"up"`.

If the "Down" key is pressed:

- ***`keyH.downPressed == true`*** *evaluates as* `true`.

- *The `direction` is set to `"down"`.*

The same applies for "Left" and "Right" keys.

If multiple keys are pressed simultaneously:

- The **first condition** in the sequence will take priority (e. g., "Up" over "Down").

```
//CHECK TILE COLLISION
collsionOn = false;
gp.cChecker.checkTile( entity: this);

//CHECK OBJECT COLLISION
int objIndex = gp.cChecker.checkObject( entity: this,  player: true);
pickUpItem(objIndex);

//IF COLLISION IS FALSE, PLAYER CAN MOVE
 if (keyH.upPressed == true || keyH.downPressed == true ||
         keyH.leftPressed == true || keyH.rightPressed == true) {

     if (collsionOn == false) {
         switch (direction) {
             case "up":
                 worldY -= speed;
                 break;
             case "down":
                 worldY += speed;
                 break;
             case "left":
                 worldX -= speed;
                 break;
             case "right":
                 worldX += speed;
                 break;
         }
     }
```

collsionOn = false;

The variable collsionOn is set to false prior to examining for tile collisions. This flag will show whether or not there has been a collision with a tile. By default, it assumes there's no collision.

gp.cChecker.checkTile(this);:
The method checkTile(this) is called to check if the player (represented by this) has collided with any tile on the map (e.g., a wall, obstacle, or boundary).

- gp likely refers to the game engine or game world object.
- cChecker is a component of the game engine responsible for checking collisions.
- This function will update the collsionOn variable based on whether a collision with a tile occurred. If the player has collided with a tile, collsionOn will be set to true.

gp.cChecker.checkObject(this, true);:
This method checks if the player has collided with any interactive objects in the game world (e.g., items, NPCs, or triggers).

• The true argument may describe the kind of object to check for or if the check should take into account whether the player is permitted to interact with the item.

• If there is an object that the player has collided with, its index is probably objIndex. It may give -1 if no objects are collided with.
• pickUpItem(objIndex);: This method handles the item pickup logic in the event that the player collides with an object.
• It gives the method the objIndex, which probably enables the player to pick up the object (or, depending on the object, take another action).

if (keyH.upPressed == true || keyH.downPressed == true ||

keyH.leftPressed == true || keyH.rightPressed == true) {

This line checks if any of the directional keys (up, down, left, or right) are pressed. The variable keyH likely refers to a handler or manager that tracks key inputs from the player.

The upPressed, downPressed, leftPressed, and rightPressed are boolean flags that indicate whether the respective keys are being pressed.

if (collsionOn == false) {

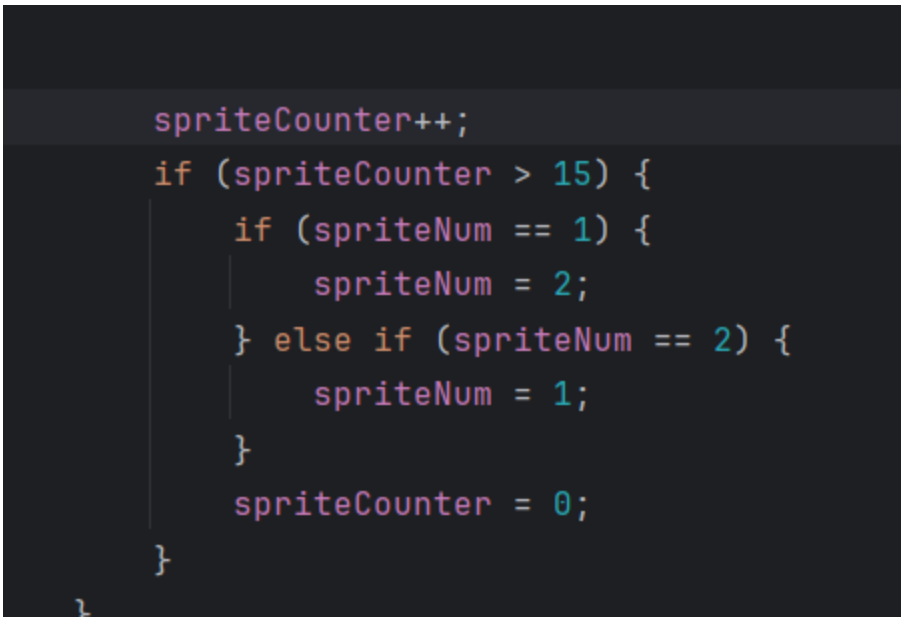
```
switch (direction) {
  case "up":
    worldY -= speed;
    break;
  case "down":
    worldY += speed;
    break;
```

```
    case "left":
      worldX -= speed;
      break;
    case "right":
      worldX += speed;
      break;
  }
}
```

if (collsionOn == false):
This condition ensures that the player can only move if there are no collisions. If collsionOn is false (meaning there is no collision with a tile), the player is allowed to move. If a collision occurred, movement is prevented.

switch (direction):
The direction variable holds the current direction the player is facing or attempting to move in. Based on this direction, the player's position is updated:

- **case "up": worldY -= speed;:**
  If the direction is "up", the player's Y-coordinate (worldY) is decreased by the speed value, moving the player upwards.
- **case "down": worldY += speed;:**
  If the direction is "down", the player's Y-coordinate is increased, moving the player downward.
- **case "left": worldX -= speed;:**
  If the direction is "left", the player's X-coordinate (worldX) is decreased, moving the player to the left.
- **case "right": worldX += speed;:**
  If the direction is "right", the player's X-coordinate is increased, moving the player to the right.



spriteCounter++;

- *This line increments the spriteCounter by 1 each time the code is executed. This counter tracks how many "ticks" or game frames have passed.*

if (spriteCounter > 15) {

- *This checks if the spriteCounter exceeds 15. When it does, it triggers a sprite change. The number 15 defines the frame rate of the animation, so the sprite will change after 16 frames (counting from 0 to 15).*

if (spriteNum == 1) {

- *If the current sprite number (spriteNum) is 1, the sprite is changed to 2.*

spriteNum = 2;

- *This sets spriteNum to 2, changing the character's sprite to the second one.*

else if (spriteNum == 2) {

- *If spriteNum is already 2, the sprite is changed back to 1.*

spriteNum = 1;

- *This sets spriteNum to 1, changing the character's sprite to the first one.*

spriteCounter = 0;

- *After the sprite change, the spriteCounter is reset to 0 to start counting for the next animation cycle.*

```
public void pickUpItem(int i){  1 usage
```

- *The index i identifies the item being picked up.*

- *The method performs various actions, such as updating the player's inventory, playing sounds, applying effects, and modifying the game world state.*

```
if (i!=999){
    String objName = gp.obj[i].name;
```

- if (i != 999):

- *This condition checks whether the value of i is not equal to 999. The value 999 is likely being used as a special flag or sentinel value to indicate a non-valid index or a special case.*
- *If i is not equal to 999, the block of code inside the if statement will execute.*

- String ***objName = gp.obj[i].name;***

- ***gp.obj[i]:*** *This accesses an object in the gp.obj array (likely representing a collection of game objects). Here, gp could refer to the game object, and obj could be an array of objects in the game world.*
- ***.name****: The name property refers to the name of the object at index i in the gp.obj array. This could represent the object's descriptive name, like "Health Potion" or "Key."*
- ***String objName = ...;*** *This assigns the name of the object to a string variable objName, which can then be used for further logic, such as displaying the object name in the UI, performing an action with the item, or showing a message.*

```
switch (objName){
    case "KEY":
        gp.playSE( i: 1);
        hasKey++;
        gp.obj[i] = null;
        gp.ui.showMessage( text: "You got a key! ");
        break;
    case "DOOR":

        if (hasKey > 0){
            gp.obj[i] = null;
            hasKey--;
            gp.playSE( i: 3);
            gp.ui.showMessage( text: "You opened the door!");
        }else {
            gp.ui.showMessage( text: "You need a key! ");
        }
```

switch (objName): *This statement assesses the object's name, objName, and, depending on its value, performs various operations. Here, "KEY" and "DOOR" are the two potential object names that are checked.*

Case for "KEY"

- **gp.playSE(1);:**
  This plays a sound effect (ID 1). This could be the sound of the player picking up a key.
- **hasKey++;:**
  This increments the hasKey variable, which tracks how many keys the player has collected. The player now has one more key.
- **gp.obj[i] = null;:**
  This removes the key object from the game world by setting the object at index i in the gp.obj array to null. This ensures the key is no longer present in the game after being picked up.
- **gp.ui.showMessage("You got a key! ");:**
  This displays a message to the player (via the UI) confirming that they picked up a key.

Case for "DOOR"

if (hasKey > 0):
This checks if the player has at least one key (hasKey > 0). If the player has a key, they can open the door.

- **gp.obj[i] = null;:**
  The door object is removed from the game world, as the player has opened it.

- **hasKey--;:**
  The player's key count is decremented because they used one key to open the door.

- **gp.playSE(3);:**
  A sound effect with ID 3 is played (likely a sound for the door opening).

- **gp.ui.showMessage("You opened the door!");:**
  A message is shown to the player, indicating that the door has been opened.

  else:
  If the player doesn't have any keys (hasKey == 0), a message is shown telling the player that they need a key to open the door.

- **gp.ui.showMessage("You need a key! ");:**
  A message is shown informing the player that they cannot open the door because they don't have a key.

```
                    break;
                case "BOOTS":
                    gp.playSE( i: 2);
                    speed += 1;
                    gp.obj[i] = null;
                    gp.ui.showMessage( text: "Speed up! ");
                    break;
                case "CHEST":
                    gp.playSE( i: 4);
                    gp.obj[i] = null;
                    gp.ui.gameFinished = true;
                    gp.stopMusic();
                    break;
            }
        }

    }
```

Case for "BOOTS"

gp.playSE(2);:
This plays a sound effect with ID 2, likely representing the player acquiring boots or a speed boost.

speed += 1;:
This increases the player's speed by 1, meaning the player will move faster after picking up the boots.

gp.obj[i] = null;:
This removes the boots object from the game world by setting the object at index i in the gp.obj array to null. This ensures the boots no longer appear in the game after being picked up.

gp.ui.showMessage("Speed up! ");:
A message is displayed to the player, confirming that their speed has increased.

Case for "CHEST":

```java
public void draw(Graphics2D g2){  1 usage

    /* g2.setColor(Color.white);
     g2.fillRect(x, y, gp.titleSize, gp.titleSize);
     */


     BufferedImage image = null;

     switch (direction) {
         case "up":
             if (spriteNum == 1) {
                 image = up1;
             }
             if (spriteNum == 2) {
                 image = up2;
             }

             break;
         case "down":
             if (spriteNum == 1) {
                 image = down1;
             }
             if (spriteNum == 2) {
                 image = down2;
             }
             break;
```

```
            case "left":
                if (spriteNum == 1) {
                    image = left1;
                }
                if (spriteNum == 2) {
                    image = left2;
                }
                break;
            case "right":
                if(spriteNum == 1){
                image = right1;
                }
                if(spriteNum == 2){
                    image = right2;
                }
                break;

        }
        g2.drawImage(image, screenX, screenY, gp.titleSize, gp.titleSize,  observer: null);

    }
```

public void draw(Graphics2D g2) {

It takes a Graphics2D object (g2) as an argument, which is used for drawing 2D graphics on the screen.

BufferedImage image = null;

- *A BufferedImage object (image) is declared to hold the current sprite image that will be drawn on the screen.*
- *if the player is moving up, it selects either up1 or up2 based on the current sprite number (spriteNum).*

Handling Each Direction

- *If the player is moving down, it selects either down1 or down2.*
- *If the player is moving left, it selects either left1 or left2*
- *If the player is moving right, it selects either right1 or right2.*

g2.drawImage(image, screenX, screenY, gp.titleSize, gp.titleSize, null);

image: The image to be drawn.

screenX and screenY: The coordinates on the screen where the image will be drawn.

gp.titleSize: The width and height of the image (it is likely square, with both width and height set to the same value).

null: This could be used for image observers, but it's not needed in this case.

2. AssetSetter

```
package main;

import object.OBJ_BOOTS;
import object.OBJ_CHEST;
import object.OBJ_DOOR;
import object.OBJ_KEY;

public class AssetSetter {  2 usages

    GamePanel gp;  41 usages

    public  AssetSetter(GamePanel gp) { this.gp = gp; }

    public  void setObject(){  1 usage
        gp.obj[0] = new OBJ_KEY();
        gp.obj[0].worldX = 23 * gp.titleSize;
        gp.obj[0].worldY = 7 * gp.titleSize;

        gp.obj[1] = new OBJ_KEY();
        gp.obj[1].worldX = 23 * gp.titleSize;
        gp.obj[1].worldY = 40 * gp.titleSize;

        gp.obj[2] = new OBJ_KEY();
        gp.obj[2].worldX = 37 * gp.titleSize;
        gp.obj[2].worldY = 7 * gp.titleSize;

        gp.obj[3] = new OBJ_DOOR();
        gp.obj[3].worldX = 10 * gp.titleSize;
        gp.obj[3].worldY = 11* gp.titleSize;
```

```
        gp.obj[4] = new OBJ_DOOR();
        gp.obj[4].worldX = 8 * gp.titleSize;
        gp.obj[4].worldY = 28 * gp.titleSize;

        gp.obj[5] = new OBJ_DOOR();
        gp.obj[5].worldX = 12 * gp.titleSize;
        gp.obj[5].worldY = 22* gp.titleSize;

        gp.obj[6] = new OBJ_CHEST();
        gp.obj[6].worldX = 10 * gp.titleSize;
        gp.obj[6].worldY = 7 * gp.titleSize;

        gp.obj[7] = new OBJ_BOOTS();
        gp.obj[7].worldX = 37 * gp.titleSize;
        gp.obj[7].worldY = 42 * gp.titleSize;
    }
}
```

package main;

import object.OBJ_BOOTS;

import object.OBJ_CHEST;

import object.OBJ_DOOR;

import object.OBJ_KEY;

- package main;: This specifies the package the AssetSetter class belongs to, which is likely the main package of the game.

 import object.*;: These imports bring in the object classes (OBJ_BOOTS, OBJ_CHEST, OBJ_DOOR, OBJ_KEY) so they can be instantiated and used in this class.

public class AssetSetter {

-*Declares the AssetSetter class, which is responsible for setting objects on the game map.*

GamePanel gp;

-*A reference to the GamePanel class (gp). This object contains game-related information and assets, such as an array of objects (gp.obj) and the tile size (gp.titleSize).*

public AssetSetter(GamePanel gp) {

this.gp = gp;

}

-*A constructor that takes a GamePanel instance (gp) as a parameter and assigns it to the gp field of this class.*

 This allows the AssetSetter class to access gp's properties, such as the array of objects and game configurations.

public void setObject() {

This method initializes and places various game objects (OBJ_KEY, OBJ_DOOR, OBJ_CHEST, OBJ_BOOTS) at specific positions on the game map.

gp.obj[0] = new OBJ_KEY();

gp.obj[0].worldX = 23 * gp.titleSize;

gp.obj[0].worldY = 7 * gp.titleSize;

- gp.obj[0] = new OBJ_KEY();

*Creates a new OBJ_KEY object and assigns it to the first index of the gp.obj array***.**

 *gp.obj[0].worldX = 23 * gp.titleSize;: Sets the worldX coordinate of the key to 23 * gp.titleSize. This positions the key horizontally on the map, where gp.titleSize likely represents the width of a single tile.*

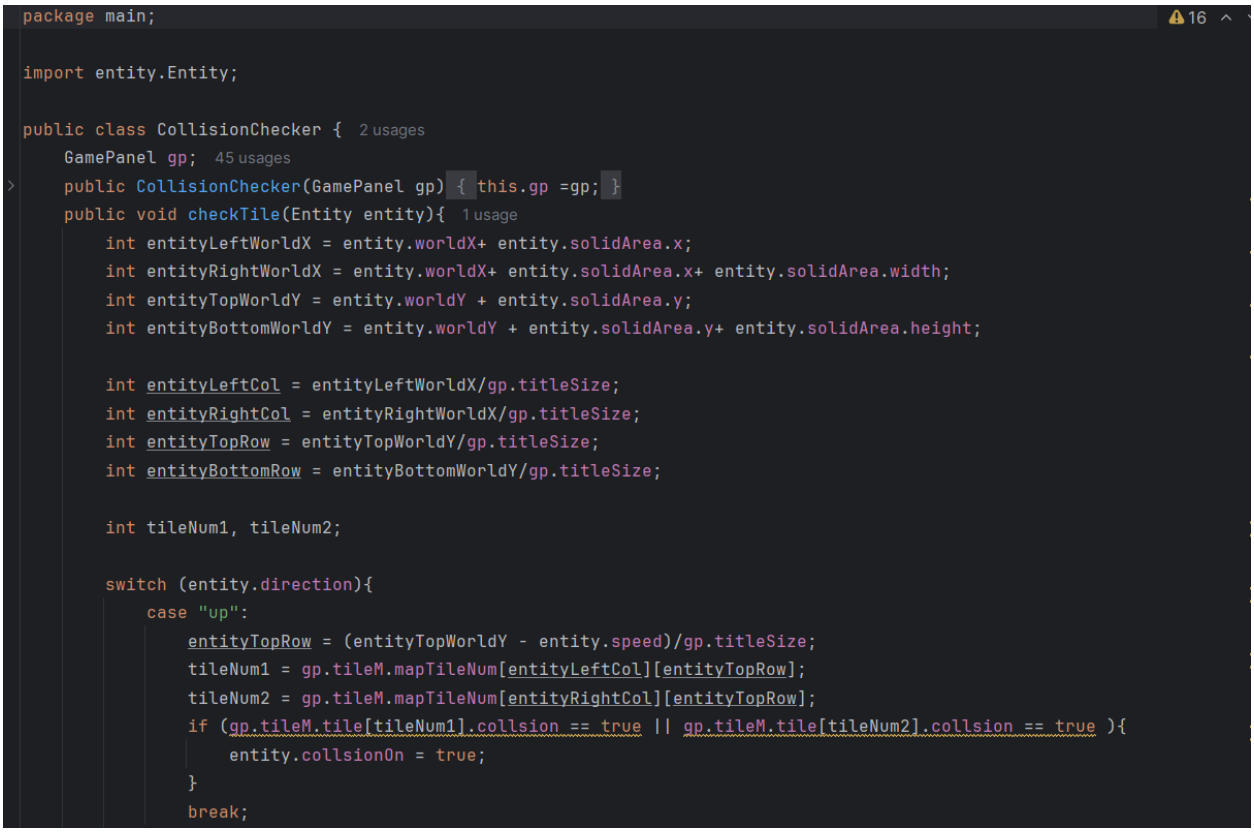
 *gp.obj[0].worldY = 7 * gp.titleSize;: Sets the worldY coordinate of the key to 7 * gp.titleSize, positioning it vertically.*

- *Keys (OBJ_KEY):*

- *Three keys are placed at specific map coordinates.*

- *Doors (OBJ_DOOR):*
  - *Three doors are placed, each corresponding to specific map coordinates.*
- *Chest (OBJ_CHEST):*
  - *A single chest is placed at a specific map coordinate, potentially marking the game's goal or treasure.*
- *Boots (OBJ_BOOTS):*
  - *A pair of boots is placed, likely providing a speed boost to the player upon collection.*

3. CollisionChecker



```
package main;

import entity.Entity;

public class CollisionChecker {  2 usages
    GamePanel gp;  45 usages
    public CollisionChecker(GamePanel gp) { this.gp =gp; }
    public void checkTile(Entity entity){  1 usage
        int entityLeftWorldX = entity.worldX+ entity.solidArea.x;
        int entityRightWorldX = entity.worldX+ entity.solidArea.x+ entity.solidArea.width;
        int entityTopWorldY = entity.worldY + entity.solidArea.y;
        int entityBottomWorldY = entity.worldY + entity.solidArea.y+ entity.solidArea.height;

        int entityLeftCol = entityLeftWorldX/gp.titleSize;
        int entityRightCol = entityRightWorldX/gp.titleSize;
        int entityTopRow = entityTopWorldY/gp.titleSize;
        int entityBottomRow = entityBottomWorldY/gp.titleSize;

        int tileNum1, tileNum2;

        switch (entity.direction){
            case "up":
                entityTopRow = (entityTopWorldY - entity.speed)/gp.titleSize;
                tileNum1 = gp.tileM.mapTileNum[entityLeftCol][entityTopRow];
                tileNum2 = gp.tileM.mapTileNum[entityRightCol][entityTopRow];
                if (gp.tileM.tile[tileNum1].collsion == true || gp.tileM.tile[tileNum2].collsion == true ){
                    entity.collsionOn = true;
                }
                break;
```

package main;

import entity.Entity;

- *package main;: The CollisionChecker class is part of the main package.*
- *import entity.Entity;: It uses the Entity class, representing movable objects in the game world.*

GamePanel gp;

- *A reference to the GamePanel class (gp), which contains information about the game map, tiles, and entities.*

  public CollisionChecker(GamePanel gp) {
    this.gp = gp;
  }
- *The constructor initializes the CollisionChecker class with a reference to the GamePanel object, giving access to tile and map data*

  public void checkTile(Entity entity) {
- *This method checks if the given entity will collide with a tile in the direction it's moving*

  .int entityLeftWorldX = entity.worldX + entity.solidArea.x;
  int entityRightWorldX = entity.worldX + entity.solidArea.x + entity.solidArea.width;
  int entityTopWorldY = entity.worldY + entity.solidArea.y;
  int entityBottomWorldY = entity.worldY + entity.solidArea.y + entity.solidArea.height*;*
- *These calculate the exact positions of the entity's edges in the game world based on its position (worldX, worldY) and its collision box (solidArea).*

  int entityLeftCol = entityLeftWorldX / gp.titleSize;
  int entityRightCol = entityRightWorldX / gp.titleSize;
  int entityTopRow = entityTopWorldY / gp.titleSize;
  int entityBottomRow = entityBottomWorldY / gp.titleSize*;*
  Converts the entity's edge coordinates from world units to tile units by dividing by gp.titleSize, which is the size of a tile.
  entityTopRow = (entityTopWorldY - entity.speed) / gp.titleSize;
  tileNum1 = gp.tileM.mapTileNum[entityLeftCol][entityTopRow];
  tileNum2 = gp.tileM.mapTileNum[entityRightCol][entityTopRow];
  if (gp.tileM.tile[tileNum1].collsion == true || gp.tileM.tile[tileNum2].collsion == true) {
    entity.collsionOn = true;
  }
- *The entity's top row is recalculated based on its current speed.*
- *The tiles at the top-left (tileNum1) and top-right (tileNum2) corners of the entity are checked.*
- *If either tile has collision == true, the entity's collsionOn flag is set to true.*

```
entityBottomRow = (entityBottomWorldY + entity.speed) / gp.titleSize;
tileNum1 = gp.tileM.mapTileNum[entityLeftCol][entityBottomRow];
tileNum2 = gp.tileM.mapTileNum[entityRightCol][entityBottomRow];
if (gp.tileM.tile[tileNum1].collsion == true || gp.tileM.tile[tileNum2].collsion == true) {
  entity.collsionOn = true;
}
```

- *The entity's bottom row is recalculated based on its speed.*
- *Checks the bottom-left and bottom-right tiles for collisions.*

```
entityLeftCol = (entityLeftWorldX - entity.speed) / gp.titleSize;
tileNum1 = gp.tileM.mapTileNum[entityLeftCol][entityTopRow];
tileNum2 = gp.tileM.mapTileNum[entityLeftCol][entityBottomRow];
if (gp.tileM.tile[tileNum1].collsion == true || gp.tileM.tile[tileNum2].collsion == true) {
  entity.collsionOn = true;
}
```

- *The leftmost column is recalculated based on the entity's speed.*
- *Checks the top-left and bottom-left tiles for collisions.*

```
entityRightCol = (entityRightWorldX + entity.speed) / gp.titleSize;
tileNum1 = gp.tileM.mapTileNum[entityRightCol][entityTopRow];
tileNum2 = gp.tileM.mapTileNum[entityRightCol][entityBottomRow];
if (gp.tileM.tile[tileNum1].collsion == true ||
gp.tileM.tile[tileNum2].collsion == true) {
    entity.collsionOn = true;
}
```

- The rightmost column is recalculated based on the entity's speed.
- Checks the top-right and bottom-right tiles for collisions.

gp.tileM.mapTileNum
gp.tileM.tile

- *gp.tileM.mapTileNum: A 2D array representing the map, where each value corresponds to a specific tile.*
- *gp.tileM.tile: An array of tile objects, where each tile has a collision property indicating if it's solid.*

```java
public class CollisionChecker {  2 usages
    public void checkTile(Entity entity){  1 usage
            case "down":
                entityBottomRow = (entityBottomWorldY + entity.speed)/gp.titleSize;
                tileNum1 = gp.tileM.mapTileNum[entityLeftCol][entityBottomRow];
                tileNum2 = gp.tileM.mapTileNum[entityRightCol][entityBottomRow];
                if (gp.tileM.tile[tileNum1].collsion == true || gp.tileM.tile[tileNum2].collsion == true ){
                    entity.collsionOn = true;
                }
                break;
            case "left":
                entityLeftCol = (entityLeftWorldX - entity.speed)/gp.titleSize;
                tileNum1 = gp.tileM.mapTileNum[entityLeftCol][entityTopRow];
                tileNum2 = gp.tileM.mapTileNum[entityLeftCol][entityBottomRow];
                if (gp.tileM.tile[tileNum1].collsion == true || gp.tileM.tile[tileNum2].collsion == true ){
                    entity.collsionOn = true;
                }
                break;
            case "right":
                entityRightCol = (entityRightWorldX + entity.speed)/gp.titleSize;
                tileNum1 = gp.tileM.mapTileNum[entityRightCol][entityTopRow];
                tileNum2 = gp.tileM.mapTileNum[entityRightCol][entityBottomRow];
                if (gp.tileM.tile[tileNum1].collsion == true || gp.tileM.tile[tileNum2].collsion == true ){
                    entity.collsionOn = true;
                }
                break;
        }
    }
```

```
public  int checkObject(Entity entity, boolean player){  1 usage
    int index = 999;

    for (int i = 0; i < gp.obj.length; i++) {
        if (gp.obj[i] != null) {
            //get entity's solid area position
            entity.solidArea.x = entity.worldX + entity.solidArea.x;
            entity.solidArea.y = entity.worldY + entity.solidArea.y;

            //get the object's solid area position
            gp.obj[i].solidArea.x = gp.obj[i].worldX + gp.obj[i].solidArea.x;
            gp.obj[i].solidArea.y = gp.obj[i].worldY + gp.obj[i].solidArea.y;


            switch (entity.direction) {
                case "up":
                    entity.solidArea.y -= entity.speed;
                    if (entity.solidArea.intersects(gp.obj[i].solidArea)) {
                        if (gp.obj[i].collision == true){
                            entity.collsionOn = true;
                        }if (player == true){
                            index = i;
                        }
                    }
                    break;
```

```
                case "down":
                    entity.solidArea.y += entity.speed;
                    if (entity.solidArea.intersects(gp.obj[i].solidArea)) {
                        if (gp.obj[i].collision == true){
                            entity.collsionOn = true;
                        }if (player == true){
                            index = i;
                        }
                    }
                    break;
                case "left":
                    entity.solidArea.x -= entity.speed;
                    if (entity.solidArea.intersects(gp.obj[i].solidArea)) {
                        if (gp.obj[i].collision == true){
                            entity.collsionOn = true;
                        }if (player == true){
                            index = i;
                        }

                    }
                    break;
                case "right":
                    entity.solidArea.x += entity.speed;
                    if (entity.solidArea.intersects(gp.obj[i].solidArea)) {
                        if (gp.obj[i].collision == true){
                            entity.collsionOn = true;
                        }if (player == true){
                            index = i;
```

public int checkObject(Entity entity, boolean player) {

int index = 999;

Entity entity: The moving entity whose collision needs to be checked.

boolean player: Indicates if the entity is the player. If true, additional actions are taken (e.g., returning the index of the collided object).

for (int i = 0; i < gp.obj.length; i++) {

if (gp.obj[i] != null) {

Condition: gp.obj[i] != null ensures the code only processes valid objects and skips any null entries in the array.

entity.solidArea.x = entity.worldX + entity.solidArea.x;

entity.solidArea.y = entity.worldY + entity.solidArea.y;

Purpose: Updates the solidArea position of the entity by combining its world coordinates (worldX, worldY) with its current solidArea offset.

Effect: This represents the entity's solid area in the global coordinate system.

```
gp.obj[i].solidArea.x = gp.obj[i].worldX + gp.obj[i].solidArea.x;

gp.obj[i].solidArea.y = gp.obj[i].worldY + gp.obj[i].solidArea.y;
```

- *Purpose: Updates the solidArea position of the current object in the same way as the entity. This aligns the object's solidArea to the global coordinate system.*

```
case "up":
  entity.solidArea.y -= entity.speed;
```

- Purpose: Simulates moving the entity upward by reducing its `solidArea`'s `y` coordinate by the entity's speed.

```
if (entity.solidArea.intersects(gp.obj[i].solidArea)) {
```

- *Purpose: Checks if the entity's adjusted solidArea intersects with the object's solidArea. This determines if a collision occurs.*

```
if (gp.obj[i].collision == true) {
  entity.collsionOn = true;
}
```

- Purpose: If the object's `collision` property is `true`, the entity's `collsionOn` flag is set to `true`, indicating a collision.

```
if (player == true) {
  index = i;
}
```

- Purpose: If the `player` flag is `true`, the index of the collided object is stored in the variable `index`. This allows the method to return which object the player collided with.

```
entity.solidArea.x = entity.solidAreaDefaultX;

entity.solidArea.y = entity.solidAreaDefaultY;

gp.obj[i].solidArea.x = gp.obj[i].solidAreaDefaultX;

gp.obj[i].solidArea.y = gp.obj[i].solidAreaDefaultY;
```

- *Purpose: After the collision checks, the solidArea positions are reset to their default values to avoid affecting future calculations.*

```
return index;
```

- Purpose: Returns the index of the object that was collided with. If no collision occurs, it returns `999` (default value).

4. GamePanel

```
package main;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.swing.JPanel;

import entity.Player;
import object.SuperObject;
import tile.TileManager;
```

the java.awt package, which is used for drawing and working with GUI components:

- ***Color**: Represents colors to be used in the GUI.*
- ***Dimension**: Represents the width and height of a GUI component (e.g., a panel or window).*
- ***Graphics**: Provides the base class for drawing operations, such as lines, shapes, or text.*
- ***Graphics2D**: Extends Graphics to provide advanced drawing capabilities like transformations, anti-aliasing, and complex shapes.*

  import javax.swing.JPanel;

  JPanel: A lightweight container used to organize components or perform custom painting in a Java Swing application.

  import entity.Player;

  - *The Player class likely represents the player's character in the game and contains its attributes (e.g., position, speed) and behavior (e.g., movement, actions).*

import object.SuperObject;

SuperObject: Likely a parent class for all objects in the game. It could define common properties (e.g., position, size) and methods (e.g., rendering) for game objects.

import tile.TileManager;

TileManager: Likely responsible for managing the game's tile-based map. It may handle loading, rendering, and interacting with tiles.

```
public class GamePanel extends JPanel implements Runnable{  16 usages
    //SCREEN SETTING

    final int originalTitleSize = 16;  1 usage
    final int scale = 3;  1 usage

    public final int titleSize = originalTitleSize * scale;  56 usages
    public final int maxScreenCol = 16;  1 usage
    public final int maxScreenRow = 12;  1 usage
    public final int screenWidth = titleSize * maxScreenCol;  5 usages
    public final int screenHeight = titleSize * maxScreenRow;  5 usages

    //WORLD SETTING
    public  final  int  maxWorldCol = 50;  6 usages
    public  final  int  maxWorldRow = 50;  3 usages


    //FPS
    int FPS = 60;  1 usage

    //SYSTEM
    TileManager tileM = new TileManager( gp: this);  17 usages
    KeyHandler keyH = new KeyHandler();  2 usages
    Sound music = new Sound();  4 usages
    Sound se = new Sound();  2 usages
    Thread gameThread;  4 usages
    public CollisionChecker cChecker = new CollisionChecker( gp: this);  2 usages
    public AssetSetter aSetter = new AssetSetter( gp: this);  1 usage
    public UI ui = new UI( gp: this);  6 usages
```

```
    //ENTITY AND OBJECT
    public Player player = new Player( gp: this, keyH);  27 usages
    public SuperObject obj[] = new SuperObject[10];  52 usages
```

public class GamePanel extends JPanel implements Runnable {

- *GamePanel: This class is the main panel of the game, where all rendering and game logic happen.*

- extends JPanel: Inherits from JPanel, meaning GamePanel acts as a canvas for drawing game elements and handling GUI operations.

-implements Runnable: Indicates the class supports multi-threading. The Runnable interface allows GamePanel to run its own thread for game updates and rendering.

final int originalTitleSize = 16;

final int scale = 3;

public final int titleSize = originalTitleSize * scale;

public final int maxScreenCol = 16;

public final int maxScreenRow = 12;

public final int screenWidth = titleSize * maxScreenCol;

public final int screenHeight = titleSize * maxScreenRow

originalTitleSize: The base size of a single tile in pixels.

scale: Scales the base tile size to fit modern screens.

titleSize: Final size of each tile after scaling.

maxScreenCol and maxScreenRow: Maximum number of tiles displayed horizontally and vertically on the screen.

screenWidth and screenHeight: Total width and height of the game screen in pixels, calculated by multiplying tile size by the number of tiles.

public final int maxWorldCol = 50;

public final int maxWorldRow = 50;

maxWorldCol and maxWorldRow: Represent the number of tiles in the game world horizontally and vertically. This is larger than the visible screen, indicating a scrolling or navigable world.

int FPS = 60;

60 FPS: A standard value for smooth gameplay.

TileManager tileM = new TileManager(this);

KeyHandler keyH = new KeyHandler();

Sound music = new Sound();

Sound se = new Sound();

Thread gameThread;

public CollisionChecker cChecker = new CollisionChecker(this);

public AssetSetter aSetter = new AssetSetter(this);

public UI ui = new UI(this);

TileManager tileM: Manages the game's tile map, including loading and rendering tiles.

KeyHandler keyH: Handles keyboard inputs for player movement and actions.

Sound music and Sound se: Used for playing background music (music) and sound effects (se).

Thread gameThread: The main thread for running the game loop (update and render cycles).

CollisionChecker cChecker: Handles collision detection between the player, objects, and tiles.

AssetSetter aSetter: Initializes and places game objects, like enemies or items, in the game world.

UI ui: Manages the game's user interface, such as health bars or score displays.

public Player player = new Player(this, keyH);

public SuperObject obj[] = new SuperObject[10];

Player player: Represents the main player character. It is initialized with a reference to the GamePanel and KeyHandler for movement and interactions.

SuperObject[] obj: An array to store up to 10 objects (SuperObject) in the game, such as items, obstacles, or collectibles.

```
public GamePanel(){  1 usage

    this.setPreferredSize(new Dimension(screenWidth, screenHeight));
    this.setBackground(Color.BLACK);
    this.setDoubleBuffered(true);
    this.addKeyListener(keyH);
    this.setFocusable(true);
}

public void setupGame(){  1 usage
    aSetter.setObject();

    playMusic( i: 0 );
}

public void startGameThread(){  1 usage

    gameThread = new Thread( task: this);
    gameThread.start();
}
```

public GamePanel() {

this.setPreferredSize(new Dimension(screenWidth, screenHeight));

this.setBackground(Color.BLACK);

this.setDoubleBuffered(true);

this.addKeyListener(keyH);

this.setFocusable(true);

this.setPreferredSize(new Dimension(screenWidth, screenHeight));

- *Sets the preferred size of the GamePanel to match the screenWidth and screenHeight defined earlier.*
- *Dimension: Specifies the width and height in pixels.*

this.setBackground(Color.BLACK);

- *Sets the background color of the GamePanel to black. This will be the default color before anything is drawn on it.*

this.setDoubleBuffered(true);

- *Enables double buffering for smoother rendering.*
- *Double buffering: Draws graphics in an off-screen buffer before displaying them to avoid flickering.*

this.addKeyListener(keyH);

- *Adds the KeyHandler object (keyH) as a listener for keyboard inputs. This allows the game to respond to player actions like movement or interactions.*

this.setFocusable(true);

- *Ensures the GamePanel can receive keyboard focus, making it active for capturing key events.*

public void setupGame() {

aSetter.setObject();

playMusic(0);

}

aSetter.setObject();

- *Calls the setObject() method from the AssetSetter class.*
- *Purpose: Places game objects (e.g., items, obstacles, enemies) in their initial positions within the game world.*

playMusic(0);

- *Starts playing background music. The argument 0 likely refers to a specific track or sound file in the Sound class.*

public void startGameThread() {

gameThread = new Thread(this);

gameThread.start();

}

gameThread = new Thread(this);

- *Creates a new thread for the game.*
- *this: Refers to the current GamePanel instance, which implements Runnable. This means the thread will execute the run() method defined in GamePanel.*

gameThread.start();

- *Starts the thread, which invokes the run() method to begin the game loop.*

```java
public void run() {
    double drawInterval = 1000000000/FPS;
    double nextDrawnTime = System.nanoTime() + drawInterval;

    while (gameThread != null) {


        // 1 UPDATE : update information
        update();
        // 2 DRAW : draw the screen with the updated information
        repaint();



        try {
            double remainingTime = nextDrawnTime - System.nanoTime();
            remainingTime = remainingTime/1000000;

            if (remainingTime < 0 ) {
                remainingTime = 0;
            }

            Thread.sleep((long)remainingTime);

            nextDrawnTime += drawInterval;
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
```

Calculate drawInterval:

- *Determines the time each frame should take in nanoseconds, based on the target FPS (Frames Per Second).*
- ***Example**: At 60 FPS, each frame should take approximately 16.67 ms (1 second / 60 frames).*

nextDrawnTime:

- *Stores the time when the next frame should be drawn, initialized as the current time plus the interval.*

Game Loop (while):

- *Repeats as long as the game thread (gameThread) is running.*
- ***update()****: Updates the game logic (e.g., player movement, object positions).*
- ***repaint()****: Requests the paintComponent() method to redraw the game.*

Calculate and Sleep:

- *Computes the remaining time before the next frame should be drawn and pauses the thread for that duration (Thread.sleep).*
- *Ensures consistent frame timing for smooth gameplay.*

```
    }

    public void playMusic(int i){  1 usage
        music.setFile(i);
        music.play();
        music.loop();
    }

    public void stopMusic(){  1 usage

        music.stop();
    }

    public void playSE(int i){  4 usages
        se.setFile(i);
        se.play();
    }
}
```

public void update() {

player.update();

}

Updates the game state.

Player movement.

Collision detection.

Other dynamic behaviors (e.g., animations).

```
public void paintComponent(Graphics g) {
   super.paintComponent(g);
   Graphics2D g2 = (Graphics2D) g;
   // TILE
  tileM.draw(g2);
   // OBJECT
   for (int i = 0; i < obj.length; i++) {
      if (obj[i] != null) {
         obj[i].draw(g2, this);
      }
   }
   // PLAYER
   player.draw(g2);
   // UI
   ui.draw(g2);
   g2.dispose();
}
```

super.paintComponent(g):

- *Clears the screen before rendering new content.*
- *Ensures proper rendering behavior by calling the parent JPanel's paintComponent().*

Graphics2D g2:

- *Casts the Graphics object to Graphics2D for advanced rendering options.*

Draw Tiles:

- *tileM.draw(g2): Draws the background tiles of the game.*

Draw Objects:

- *Loops through the obj array and calls the draw() method for each non-null object.*

Draw Player:

- *Calls the draw() method of the player object to render the player on the screen.*

Draw UI:

- *Calls the draw() method of the ui object to render the user interface (e.g., health bar, score).*

g2.dispose():

- *Releases system resources used by the Graphics2D object.*

public void playMusic(int i) {

music.setFile(i);

music.play();

music.loop();

}

setFile(i): Loads a specific music file based on index i.

play(): Starts playing the music.

loop(): Ensures the music plays continuously in a loop

public void stopMusic() {

music.stop();

}

Stops the currently playing background music.

public void playSE(int i) {

se.setFile(i);

se.play();

}

- ***setFile(i)****: Loads a specific sound effect file based on index i.*
- ***play()****: Plays the sound effect once.*

5. KeyHandler

```java
package main;

import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

public class KeyHandler implements KeyListener {  5 usages
    public boolean upPressed, downPressed, leftPressed, rightPressed;  4 usages


    @Override
    public void keyTyped(KeyEvent e) {

    }

    @Override
    public void keyPressed(KeyEvent e) {
        int code = e.getKeyCode();
        if (code == KeyEvent.VK_W) {
            upPressed = true;
        }
        if (code == KeyEvent.VK_S) {
            downPressed = true;
        }
        if (code == KeyEvent.VK_A) {
            leftPressed = true;
        }
        if (code == KeyEvent.VK_D) {
            rightPressed = true;
        }
    }
```

```java
@Override
public void keyReleased(KeyEvent e) {
    int code = e.getKeyCode();
    if (code == KeyEvent.VK_W) {
        upPressed = false;
    }
    if (code == KeyEvent.VK_S) {
        downPressed = false;
    }
    if (code == KeyEvent.VK_A) {
        leftPressed = false;
    }
    if (code == KeyEvent.VK_D) {
        rightPressed = false;
    }
}
```

public class KeyHandler implements KeyListener {

The KeyHandler class listens for specific key events (key presses, releases, and typing) to control player movement.

public boolean upPressed, downPressed, leftPressed, rightPressed;

- *Purpose: These boolean variables indicate whether the respective movement keys (W, A, S, D) are currently pressed.*
- *Usage:*
    - *If upPressed is true, the player is moving up.*
    - *If leftPressed is true, the player is moving left, and so on.*

@Override

public void keyTyped(KeyEvent e) {

}

Purpose: This method is called when a key is typed (pressed and released quickly).

Implementation: Left empty because it's not used in this context.

```
@Override
public void keyPressed(KeyEvent e) {
  int code = e.getKeyCode();
  if (code == KeyEvent.VK_W) {
    upPressed = true;
  }
  if (code == KeyEvent.VK_S) {
    downPressed = true;
  }
  if (code == KeyEvent.VK_A) {
    leftPressed = true;
  }
  if (code == KeyEvent.VK_D) {
    rightPressed = true;
  }
}
```

int code = e.getKeyCode();

- *Retrieves the numeric key code of the pressed key (e.g., KeyEvent.VK_W for the W key).*

  Check Key Codes:

- *If the pressed key matches a specific movement key (W, A, S, or D), the corresponding boolean variable is set to true.*
- *Key Codes:*
  - *KeyEvent.VK_W: Represents the W key for upward movement.*
  - *KeyEvent.VK_S: Represents the S key for downward movement.*
  - *KeyEvent.VK_A: Represents the A key for leftward movement.*
  - *KeyEvent.VK_D: Represents the D key for rightward movement***.**

```
@Override
public void keyReleased(KeyEvent e) {
```

```
    int code = e.getKeyCode();
    if (code == KeyEvent.VK_W) {
      upPressed = false;
    }
    if (code == KeyEvent.VK_S) {
      downPressed = false;
    }
    if (code == KeyEvent.VK_A) {
      leftPressed = false;
    }
    if (code == KeyEvent.VK_D) {
      rightPressed = false;
    }
  }
```

int code = e.getKeyCode();

- *Retrieves the numeric key code of the released key.*

  Check Key Codes:

- *If the released key matches a specific movement key (W, A, S, or D), the corresponding boolean variable is set to false.*

6. Main

```java
package main;

import javax.swing.JFrame;

public class Main {
    public static void main(String[] args) {
        JFrame window = new JFrame();
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setResizable(false);
        window.setTitle("Game");

        GamePanel gamePanel = new GamePanel();
        window.add(gamePanel);

        window.pack();


        window.setLocationRelativeTo(null);
        window.setVisible(true);
        gamePanel.setupGame();
        gamePanel.startGameThread();
    }
}
```

public class Main { public static void main(String[] args) { … }

This is the main method of the program, which serves as the entry point for the application.

It creates a window (JFrame) to run the game and initializes the GamePanel for game logic and rendering.

JFrame window = new JFrame();

Creates a new instance of JFrame, which is the main window for your game.

JFrame is part of the Swing library and provides a top-level container for GUI applications

window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

- ***Purpose**: Specifies what happens when the user clicks the close button.*
- ***JFrame.EXIT_ON_CLOSE**:*
    - *Closes the application and exits the program.*

window.setResizable(false);

- ***Purpose**: Disables resizing of the game window.*
- *Prevents the player from resizing the window, which could distort the game's visuals*

window.setTitle("Game");

- ***Purpose**: Sets the title of the game window.*
- *The text "Game" appears in the title bar of the window*

GamePanel gamePanel = new GamePanel();

- ***Purpose**: Creates an instance of GamePanel, which is the main component for game logic and rendering.*
- ***GamePanel**:*
    - *Contains the game loop, rendering logic, and event handling.*

window.add(gamePanel);

- ***Purpose**: Adds the GamePanel object to the JFrame.*
- ***Why?**:*
    - *GamePanel is where all game visuals and logic are handled, so it needs to be part of the window.*

window.pack();

- ***Purpose**: Sizes the JFrame to fit the preferred size of its components.*
- ***How?**:*
    - *The GamePanel specifies its preferred size using setPreferredSize(), and pack() ensures the JFrame matches this size.*

window.setLocationRelativeTo(null);

- ***Purpose**: Centers the game window on the screen.*
- ***How?**:*
    - *Passing null centers the window relative to the screen*

window.setVisible(true);

- ***Purpose**: Makes the window visible on the screen.*
- ***Why?**:*
    - *A JFrame is not visible by default; you must explicitly set it to be visible*

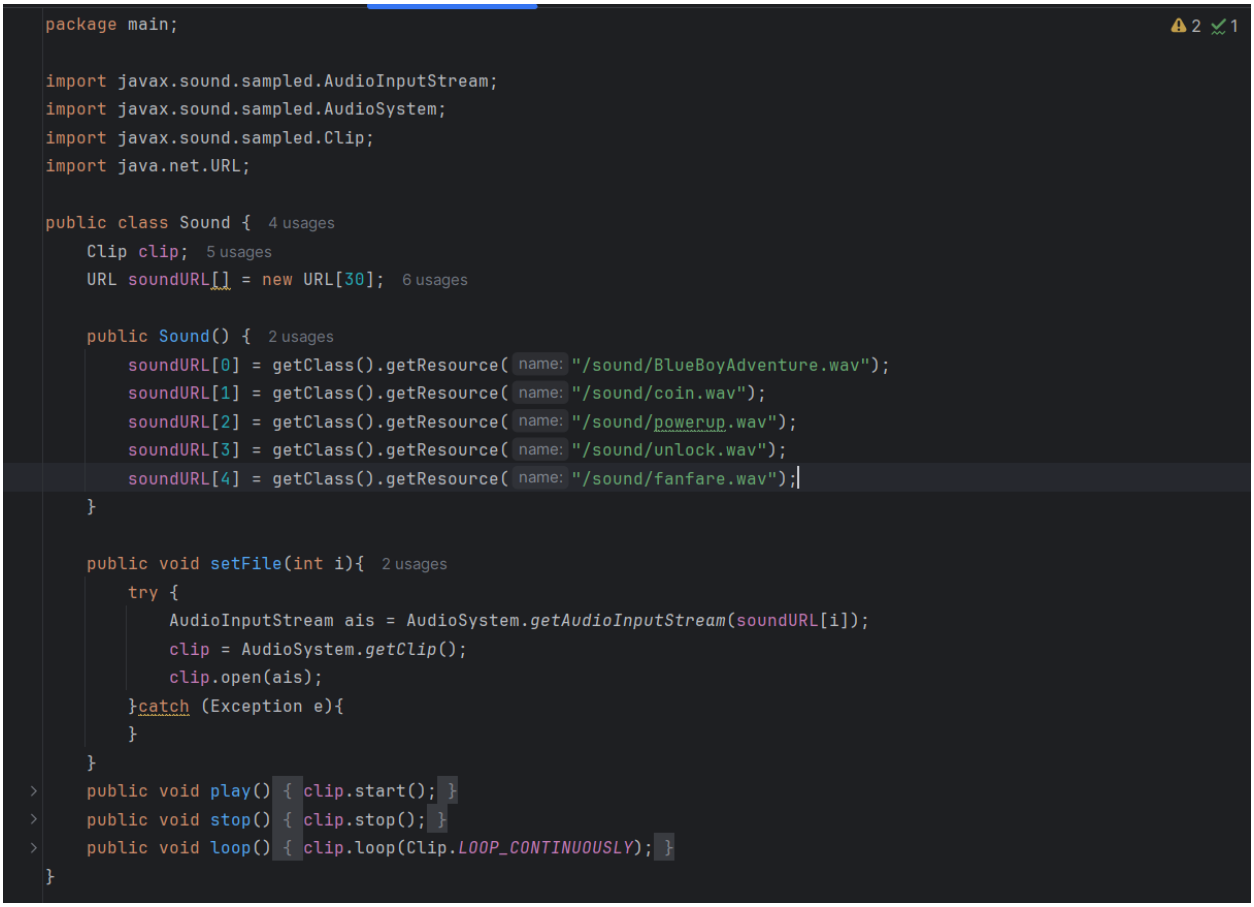
gamePanel.setupGame();

- ***Purpose**: Prepares the game by initializing objects and settings.*
- ***setupGame()**:*
    - *Likely sets up the initial game state, such as placing objects and starting background music*

gamePanel.startGameThread();

- ***Purpose**: Starts the game loop.*
- ***How?**:*
    - *The startGameThread() method creates a thread that runs the game loop defined in GamePanel.run()*

7. Sound



```
package main;

import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
import java.net.URL;

public class Sound {  4 usages
    Clip clip;  5 usages
    URL soundURL[] = new URL[30];  6 usages

    public Sound() {  2 usages
        soundURL[0] = getClass().getResource( name: "/sound/BlueBoyAdventure.wav");
        soundURL[1] = getClass().getResource( name: "/sound/coin.wav");
        soundURL[2] = getClass().getResource( name: "/sound/powerup.wav");
        soundURL[3] = getClass().getResource( name: "/sound/unlock.wav");
        soundURL[4] = getClass().getResource( name: "/sound/fanfare.wav");
    }

    public void setFile(int i){  2 usages
        try {
            AudioInputStream ais = AudioSystem.getAudioInputStream(soundURL[i]);
            clip = AudioSystem.getClip();
            clip.open(ais);
        }catch (Exception e){
        }
    }
    public void play() { clip.start(); }
    public void stop() { clip.stop(); }
    public void loop() { clip.loop(Clip.LOOP_CONTINUOUSLY); }
}
```

public class Sound {

- ***Purpose**: Defines a class to handle sound effects and music for the game.*
- *The class provides methods to load, play, stop, and loop sound files.*

Clip clip;

- ***Purpose**: Represents the audio clip currently being played.*
- ***Clip**: A part of the javax.sound.sampled package, it allows for preloading audio data and controlling playback (start, stop, loop).*

URL soundURL[] = new URL[30];

- ***Purpose**: Holds URLs of sound files.*
- ***Why an Array?**: To store multiple sound file paths for different game events (e.g., background music, sound effects).*

public Sound() { soundURL[0] = getClass().getResource("/sound/BlueBoyAdventure.wav"); soundURL[1] = getClass().getResource("/sound/coin.wav"); soundURL[2] = getClass().getResource("/sound/powerup.wav"); soundURL[3] =

getClass().getResource("/sound/unlock.wav"); soundURL[4] = getClass().getResource("/sound/fanfare.wav"); }

getClass().getResource(String path):

- *Retrieves the file located at the specified path.*
- *Assumes the files are stored in the /sound directory of the projec*

```
public void setFile(int i){
  try {
    AudioInputStream ais = AudioSystem.getAudioInputStream(soundURL[i]);
    clip = AudioSystem.getClip();
    clip.open(ais);
  } catch (Exception e) {
  }
}
```

AudioSystem.getAudioInputStream(soundURL[i]):

- ***Converts the sound file at soundURL[i] into an AudioInputStream.***

clip = AudioSystem.getClip();

- ***Creates a new Clip instance for audio playback.***

clip.open(ais);

- ***Loads the audio data from the AudioInputStream into the Clip.***

Error Handling:

- ***If an error occurs (e.g., invalid file path or unsupported audio format), it's caught by the catch block (left empty here).***

```
public void play(){
  clip.start();
}
```

- **Purpose**: Starts playing the loaded sound.
- `clip.start()`:
  - Begins playback from the current position in the audio clip.

```
public void stop(){
  clip.stop();
```

- *clip.stop():*
  - *Halts playback immediately but retains the current position in the clip.*

```
public void loop(){
  clip.loop(Clip.LOOP_CONTINUOUSLY);
}
```

Purpose: Loops the audio clip continuously.

clip.loop(Clip.LOOP_CONTINUOUSLY):

- *Sets the clip to replay continuously until explicitly stopped*

8.UI

```java
package main;

import object.OBJ_KEY;

import java.awt.*;
import java.awt.image.BufferedImage;
import java.awt.Font;
import java.text.DecimalFormat;
```

import object.OBJ_KEY; AWT (Abstract Window Toolkit) is a Java library for creating graphical user interfaces (GUIs).

Examples: Color, Graphics, Dimension, Rectangle, etc.

import java.awt.*; *AWT (Abstract Window Toolkit) is a Java library for creating graphical user interfaces (GUIs). Examples: Color, Graphics, Dimension, Rectangle, etc.*

import java.awt.image.BufferedImage; *BufferedImage is used to represent an image with an accessible buffer of image data*

import java.awt.Font; *Font is used to define the style and size of text in GUI applications.*

import java.text.DecimalFormat; *DecimalFormat is used to format decimal numbers to specific patterns (e.g., rounding, adding commas, etc.).*

```java
public class UI {  2 usages
    GamePanel gp;  19 usages
    Font font, arial_80B;  3 usages
    BufferedImage keyImage;  2 usages
    public boolean messageOn = false;  3 usages
    public String message = "";  2 usages
    int messageCounter = 0;  3 usages
    public boolean gameFinished = false;  2 usages

    double playTime;  3 usages
    DecimalFormat decimalFormat = new DecimalFormat( pattern: "#0.00");  2 usages
```

public class UI {: UI: *A class for managing the user interface (UI) of a game*

GamePanel gp;

- *A reference to the GamePanel class, likely responsible for game logic and rendering.*
- *UI interacts with GamePanel to display relevant game information.*

Font font, arial_80B;

- *font: A normal 40-point Arial font (PLAIN).*
- *arial_80B: A bold 80-point Arial font (BOLD).*

BufferedImage keyImage;

- *Holds an image of a key, likely used for rendering in the UI (e.g., inventory or HUD).*

boolean messageOn = false;

- *Indicates whether an on-screen message (e.g., notifications) should be displayed.*

String message = "";

- *Holds the text of the message to display.*

int messageCounter = 0;

- *Tracks how long a message is shown (probably for timing display duration).*

boolean gameFinished = false;

- *Tracks if the game is completed, which may trigger end screens or final messages.*

double playTime;

- *Tracks the total playtime.*

DecimalFormat decimalFormat = new DecimalFormat("#0.00");

- *Formats the playtime to two decimal places (e.g., 12.34 seconds*

UI(GamePanel gp)

- *A constructor that accepts a GamePanel object and links it to the UI class (this.gp = gp).*

font = new Font("Arial", Font.PLAIN, 40);

- *Initializes a 40-point Arial font for normal UI elements.*

arial_80B = new Font("Arial", Font.BOLD, 80);

- *Creates an 80-point bold Arial font, likely for important headers or game-over screens.*

OBJ_KEY key = new OBJ_KEY();

- *Instantiates an OBJ_KEY object.*
- *This could represent a collectible item, and OBJ_KEY likely contains an image resource.*

keyImage = key.image;

- *Extracts the key image from the OBJ_KEY object to store in keyImage for rendering*

```java
public UI(GamePanel gp) {  1 usage
    this.gp = gp;
    font = new Font( name: "Arial", Font.PLAIN,  size: 40);
    arial_80B = new Font( name: "Arial", Font.BOLD,  size: 80);
    OBJ_KEY key = new OBJ_KEY();
    keyImage = key.image;
}

public void showMessage(String text) {  4 usages

    message = text;
    messageOn = true;
}
```

```java
public void draw(Graphics2D g2) {  1 usage

    if (gameFinished == true) {

        g2.setFont(font);
        g2.setColor(Color.WHITE);

        String text;
        int textLength;
        int x;
        int y;

        text = "You found the treasure!";
        textLength = (int) g2.getFontMetrics().getStringBounds(text, g2).getWidth();
         x = gp.screenWidth/2 - textLength/2;
         y = gp.screenHeight/2 - (gp.titleSize*3);
         g2.drawString(text, x, y);

        text = "You time is: " + decimalFormat.format(playTime)+ "!";
        textLength = (int) g2.getFontMetrics().getStringBounds(text, g2).getWidth();
        x = gp.screenWidth/2 - textLength/2;
        y = gp.screenHeight/2 + (gp.titleSize);
        g2.drawString(text, x, y);
```

```
        g2.setFont(arial_80B);
        g2.setColor(Color.YELLOW);
        text = "Congratulations!!!";
        textLength = (int) g2.getFontMetrics().getStringBounds(text, g2).getWidth();
        x = gp.screenWidth/2 - textLength/2;
        y = gp.screenHeight/2 + (gp.titleSize*3);
        g2.drawString(text, x, y);

        gp.gameThread = null;

    } else {


        g2.setFont(font);
        g2.setColor(Color.WHITE);
        g2.drawImage(keyImage,  x: gp.titleSize / 2,  y: gp.titleSize / 2, gp.titleSize, gp.titleSize
        g2.drawString( str: "x " + gp.player.hasKey,  x: 74,  y: 50);

        //TIME
        playTime += (double)1/60;
        g2.drawString( str: "Time: " +decimalFormat.format(playTime),  x: gp.titleSize *11,  y: 50);

        //MESSAGE
```

```
            //MESSAGE
            if (messageOn == true) {
                g2.setFont(g2.getFont().deriveFont( size: 30F));
                g2.drawString(message,  x: gp.titleSize / 2,  y: gp.titleSize * 2);

                messageCounter++;
                if (messageCounter > 100) {
                    messageCounter = 0;
                    messageOn = false;
                }
            }
        }
    }
}
```

public void showMessage(String text) {

message = text;

messageOn = true;

}

- *Purpose:*

- *Displays a temporary message on the screen.*
- *The message is set by passing a string (text) to the method.*

public void draw(Graphics2D g2) {

- *Purpose:*
  - *Renders the game's UI elements (like messages, keys, and timers) on the screen.*
  - *Also handles the end-game screen when the game finishes.*

if(gameFinished == true) {

Condition*:*

- *If gameFinished is true, the game displays a "victory" screen.*
- *It shows congratulatory messages and the player's playtime.*

g2.setFont(font);

g2.setColor(Color.WHITE);

- *Sets the font to the normal font (Arial 40 PLAIN) and color to white*

text = "You found the treasure!";

textLength = (int) g2.getFontMetrics().getStringBounds(text, g2).getWidth();

x = gp.screenWidth / 2 - textLength / 2;

y = gp.screenHeight / 2 - (gp.titleSize * 3);

g2.drawString(text, x, y);

- ***Text Centering****:*
  - *The message "You found the treasure!" is horizontally centered.*
  - *g2.getFontMetrics().getStringBounds calculates the width of the string to align it.*
  - *x centers the text horizontally.*
  - *y determines the vertical position above the screen's midpoint.*

text = "Your time is: " + decimalFormat.format(playTime) + "!";

- *Formats and displays the player's total playtime.*
- ***decimalFormat*** *ensures the time is shown with two decimal places*

g2.setFont(arial_80B);

g2.setColor(Color.YELLOW);

text = "Congratulations!!!";

textLength = (int) g2.getFontMetrics().getStringBounds(text, g2).getWidth();

x = gp.screenWidth / 2 - textLength / 2;

y = gp.screenHeight / 2 + (gp.titleSize * 3);

g2.drawString(text, x, y);

- *Larger, bold yellow text is drawn to celebrate the win.*

- *Positioned below the playtime message.*

gp.gameThread = null;

- *Stops the game loop by setting the game thread to null.*
- *Effectively freezes the game, signaling its completion.*

} else {

- *If the game is not finished, the HUD continues to display gameplay elements like keys and time.*

g2.drawImage(keyImage, gp.titleSize / 2, gp.titleSize / 2, gp.titleSize, gp.titleSize, null);

g2.drawString("x " + gp.player.hasKey, 74, 50);

- *Draws the key image (keyImage) near the top-left corner.*
- *The number of keys the player has (gp.player.hasKey) is shown next to it.*

playTime += (double) 1 / 60;

g2.drawString("Time: " + decimalFormat.format(playTime), gp.titleSize * 11, 50);

- ***Timer**: Increments playTime by 1/60 every frame (assuming 60 FPS).*
- *Formats and displays the elapsed time at the top-right.*

if (messageOn == true) {

g2.setFont(g2.getFont().deriveFont(30F));

g2.drawString(message, gp.titleSize / 2, gp.titleSize * 2);

messageCounter++;

if (messageCounter > 100) {

messageCounter = 0;

messageOn = false;

}

}

- *If messageOn is true, the message is displayed at the top center.*

- ***Duration****:*
    - *messageCounter tracks how long the message stays.*
    - *After 100 frames (~1.66 seconds at 60 FPS), the message disappears.*

9. OBJECT

+SuperObject:

```java
package object;

import main.GamePanel;

import java.awt.*;
import java.awt.image.BufferedImage;

public class SuperObject {  7 usages  4 inheritors
    public BufferedImage image;  6 usages
    public String name;
    public  boolean collision = false;  5 usages
    public  int worldX, worldY;  12 usages
    public  Rectangle solidArea = new Rectangle( x: 0,  y: 0,  width: 48,  height: 48);  10 usages
    public  int solidAreaDefaultX = 0;  1 usage
    public  int solidAreaDefaultY = 0;  1 usage

    public void draw(Graphics2D g2, GamePanel gp){  1 usage
        int screenX = worldX - gp.player.worldX + gp.player.screenX;
        int screenY = worldY - gp.player.worldY + gp.player.screenY;

        if (worldX + gp.titleSize > gp.player.worldX - gp.player.screenX &&
                worldX - gp.titleSize< gp.player.worldX + gp.player.screenX &&
                worldY + gp.titleSize> gp.player.worldY - gp.player.screenY &&
                worldY - gp.titleSize< gp.player.worldY + gp.player.screenY) {

            g2.drawImage(image, screenX, screenY, gp.titleSize, gp.titleSize,  observer: null);
        }
    }
}
```

import main.GamePanel;

import java.awt.*;

import java.awt.image.BufferedImage;

- *main.GamePanel: Imports the GamePanel class, probably a key class in the game responsible for managing the game loop and rendering.*
- *java.awt.*: Provides graphical components like Graphics2D and Rectangle.*
- *java.awt.image.BufferedImage: Used to handle images for game objects.*

```
public BufferedImage image;
public String name;
public boolean collision = false;
public int worldX, worldY;
public Rectangle solidArea = new Rectangle(0, 0, 48, 48);
public int solidAreaDefaultX = 0;
public int solidAreaDefaultY = 0;
```

- **`image`**: A `BufferedImage` representing the graphical appearance of the object.
- **`name`**: A `String` holding the name of the object (for identification).

- **`collision`**: A boolean that determines if this object has collision detection enabled.
- **`worldX, worldY`**: Integer coordinates representing the object's position in the game world.
- **`solidArea`**: A `Rectangle` representing the collision boundary of the object (default size 48x48 pixels).
- **`solidAreaDefaultX, solidAreaDefaultY`**: Integers for resetting the `solidArea`'s position

```
public void draw(Graphics2D g2, GamePanel gp) {

  int screenX = worldX - gp.player.worldX + gp.player.screenX;

  int screenY = worldY - gp.player.worldY + gp.player.screenY;


  if (worldX + gp.titleSize > gp.player.worldX - gp.player.screenX &&

    worldX - gp.titleSize < gp.player.worldX + gp.player.screenX &&

    worldY + gp.titleSize > gp.player.worldY - gp.player.screenY &&

    worldY - gp.titleSize < gp.player.worldY + gp.player.screenY) {


    g2.drawImage(image, screenX, screenY, gp.titleSize, gp.titleSize, null);

  }

}
```

Calculate screen coordinates (screenX, screenY):

- *These are derived from the object's world coordinates and the player's position.*
- *Formula:*
    - *screenX = worldX - gp.player.worldX + gp.player.screenX*
    - *screenY = worldY - gp.player.worldY + gp.player.screenY*
- *This ensures that the object moves relative to the player's position.*

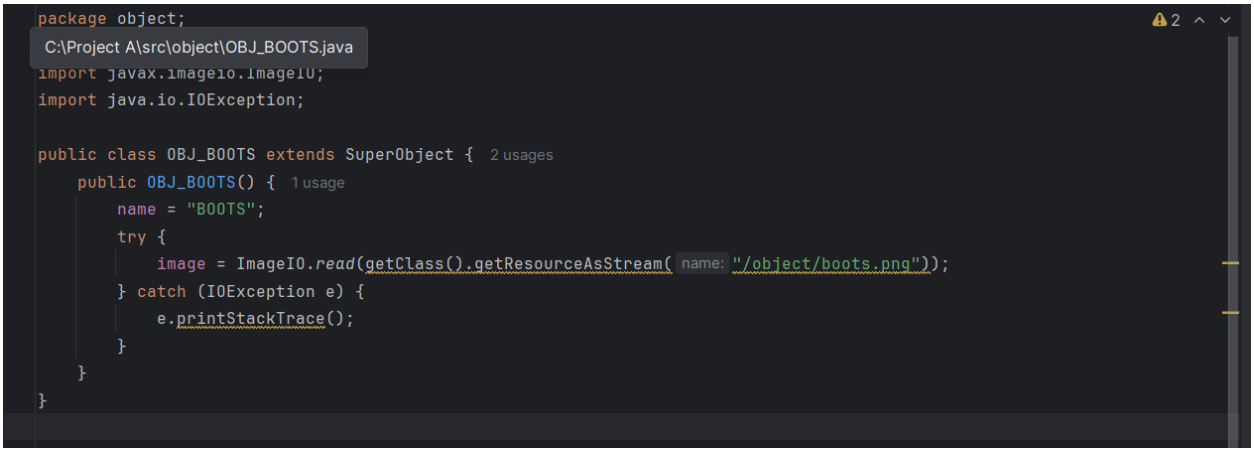
Check if the object is visible:

- *The method compares the object's world coordinates to the player's visible area (based on gp.titleSize and player's screenX and screenY).*
- *Objects outside the visible area are not drawn, optimizing performance.*

Draw the object:

- *If the object is visible, g2.drawImage is used to draw it at the calculated screen coordinates, scaled to gp.titleSize dimensions.*

+Boots:



import javax.imageio.ImageIO;

import java.io.IOException;

These imports bring in ImageIO (used for reading image files) and IOException (to handle potential errors when reading the image).

public class OBJ_BOOTS extends SuperObject {

OBJ_BOOTS is a subclass of SuperObject, meaning it inherits and can override methods and variables from SuperObject.

public OBJ_BOOTS() {

This constructor is called when an instance of OBJ_BOOTS is created.

name = "BOOTS";

The name field (likely inherited from SuperObject) is set to "BOOTS".

try {

image = ImageIO.read(getClass().getResourceAsStream("/object/boots.png"));

} catch (IOException e) {

e.printStackTrace();

}

- *The code attempts to load an image (boots.png) from the object resource folder.*
- *ImageIO.read reads the image and assigns it to the image field.*
- *If the image cannot be found or read, an IOException is caught, and the error stack trace is printed.*

+Chest:

```
package object;

import javax.imageio.ImageIO;
import java.io.IOException;

public class OBJ_CHEST extends  SuperObject{  2 usages
public OBJ_CHEST(){  1 usage
    name = "CHEST";
    try {
        image = ImageIO.read(getClass().getResourceAsStream( name: "/object/chest (OLD).png"));
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
```

+Key:

```
package object;

import javax.imageio.ImageIO;
import java.io.IOException;

public class OBJ_KEY extends SuperObject{  7 usages

    public OBJ_KEY(){  4 usages
        name = "KEY";
        try {
          image = ImageIO.read(getClass().getResourceAsStream( name: "/object/key.png"));
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

}
```

public class OBJ_KEY extends SuperObject {

- *OBJ_KEY extends SuperObject, meaning it inherits all the fields and methods of SuperObject (such as image, name, collision, etc.), and can override or modify them as needed*.

public OBJ_KEY() {

name = "KEY";

try {

image = ImageIO.read(getClass().getResourceAsStream("/object/key.png"));

} catch (IOException e) {

e.printStackTrace();

}

}

- *name = "KEY";: Sets the name field inherited from SuperObject to "KEY", identifying this object as a key.*
- *Image Loading:*
  - *ImageIO.read() is used to load the image for the object. The getClass().getResourceAsStream("/object/key.png") method is used to access the key.png image file located in the resources folder.*
  - *Exception Handling: If the image cannot be loaded (e.g., file is missing or unreadable), an IOException will be caught, and the stack trace will be printed*

+ Door:

```
package object;

import javax.imageio.ImageIO;
import java.io.IOException;

public class OBJ_DOOR extends SuperObject {  4 usages
    public OBJ_DOOR(){  3 usages
        name = "DOOR";
        try {
            image = ImageIO.read(getClass().getResourceAsStream( name: "/object/door.png"));
        } catch (IOException e) {
            e.printStackTrace();
        }
        collision = true;
    }
}
```

public class OBJ_DOOR extends SuperObject {

- *OBJ_DOOR is a subclass of SuperObject. This means it inherits fields and methods from SuperObject, such as image, name, collision, etc.*

```
public OBJ_DOOR() {
  name = "DOOR";
  try {
    image = ImageIO.read(getClass().getResourceAsStream("/object/door.png"));
  } catch (IOException e) {
    e.printStackTrace();
  }
  collision = true;
}
```

- **name = "DOOR";**: Sets the name of this object to "DOOR", identifying it as a door in the game.

- **Image Loading**:
  - The constructor uses ImageIO.read() to load the image for the door from the file path /object/door.png within the resources folder.
  - **Error Handling**: If there's an issue loading the image (e.g., the file is missing or unreadable), it will catch an IOException and print the error stack trace.
- **Collision**:
  - **collision = true;**: Sets the collision property of the door object to true, meaning the player cannot pass through the door unless specific conditions are met (like using a key or unlocking it). In games, doors often act as obstacles that require interaction for passage.

10. Tile

```
import main.GamePanel;

import javax.imageio.ImageIO;
import java.awt.*;
import java.io.*;

public class TileManager {  3 usages
```

```
public class TileManager {  3 usages
    GamePanel gp;  28 usages
    public Tile[] tile;  25 usages
    public int mapTileNum [] [];  11 usages

    public TileManager(GamePanel gp) {  1 usage
        this.gp = gp;
        tile = new Tile[10];
        mapTileNum = new int[gp.maxWorldCol][gp.maxWorldRow];
        getTilesImage();
        loadMap( filePath: "/maps/map01.txt");

    }
```

```java
public void getTilesImage() {  1 usage
    try {
        tile[0] = new Tile();
        tile[0].image = ImageIO.read(getClass().getResourceAsStream( name: "/tiles/grass.png"));

        tile[1] = new Tile();
        tile[1].image = ImageIO.read(getClass().getResourceAsStream( name: "/tiles/wall.png"));
        tile[1].collsion = true;

        tile[2] = new Tile();
        tile[2].image = ImageIO.read(getClass().getResourceAsStream( name: "/tiles/water.png"));
        tile[2].collsion = true;

        tile[3] = new Tile();
        tile[3].image = ImageIO.read(getClass().getResourceAsStream( name: "/tiles/earth.png"));

        tile[4] = new Tile();
        tile[4].image = ImageIO.read(getClass().getResourceAsStream( name: "/tiles/tree.png"));
        tile[4].collsion = true;

        tile[5] = new Tile();
        tile[5].image = ImageIO.read(getClass().getResourceAsStream( name: "/tiles/sand.png"));

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```java
public  void loadMap(String filePath){  1 usage
    try {
        InputStream is = getClass().getResourceAsStream(filePath);
        BufferedReader br = new BufferedReader(new InputStreamReader(is));

        int col = 0;
        int row = 0;
        while(col < gp.maxWorldCol && row < gp.maxWorldRow){
            String line = br.readLine();
            while (col < gp.maxWorldCol ){
                String numbers[] = line.split( regex: " ");
                int num = Integer.parseInt(numbers[col]);

                mapTileNum [col][row] = num;
                col++;
            }
            if (col == gp.maxWorldCol){
                col = 0;
                row++;
            }
        }

        br.close();
    } catch (Exception e) {

    }
```

```
ublic  void loadMap(String filePath){  1 usage

ublic  void draw(Graphics2D g2){  1 usage
    int worldCol = 0;
    int worldRow = 0;


    while(worldCol < gp.maxWorldCol && worldRow < gp.maxWorldRow){
        int tileNum = mapTileNum[worldCol][worldRow];

        int worldX = worldCol * gp.titleSize;
        int worldY = worldRow * gp.titleSize;
        int screenX = worldX - gp.player.worldX + gp.player.screenX;
        int screenY = worldY - gp.player.worldY + gp.player.screenY;

        if (worldX + gp.titleSize > gp.player.worldX - gp.player.screenX &&
            worldX - gp.titleSize< gp.player.worldX + gp.player.screenX &&
            worldY + gp.titleSize> gp.player.worldY - gp.player.screenY &&
            worldY - gp.titleSize< gp.player.worldY + gp.player.screenY) {

            g2.drawImage(tile[tileNum].image, screenX, screenY, gp.titleSize, gp.titleSize,  observer: null);
        }
        worldCol++;

        if(worldCol == gp.maxWorldCol) {
            worldCol = 0;
            worldRow++;
```

package tile;

import main.GamePanel;

import javax.imageio.ImageIO;

import java.awt.*;

import java.io.*;

- ***TileManager*** *is part of the tile package.*
- *Imports ImageIO for loading images, Graphics2D for drawing, and InputStream/BufferedReader for reading map files.*

GamePanel gp;

public Tile[] tile;

public int mapTileNum[][];

- ***gp*** *– A reference to the GamePanel, the main game component.*
- ***tile*** *– An array to hold different types of tiles (like grass, water, etc.).*
- ***mapTileNum*** *– A 2D array storing tile numbers, representing the map layout.*

```
public TileManager(GamePanel gp) {
  this.gp = gp;
  tile = new Tile[10];
  mapTileNum = new int[gp.maxWorldCol][gp.maxWorldRow];
  getTilesImage();
  loadMap("/maps/map01.txt");
}
```

- *Initializes the TileManager by:*
    1. *Assigning the GamePanel instance.*
    2. *Creating space for up to 10 tile types.*
    3. *Allocating memory for the map using the world size (maxWorldCol and maxWorldRow).*
    4. *Calls getTilesImage() to load tile images.*
    5. *Calls loadMap() to read the map from a text file.*

```
public void getTilesImage() {
  try {
    tile[0] = new Tile();
    tile[0].image =
ImageIO.read(getClass().getResourceAsStream("/tiles/grass.png"));

    tile[1] = new Tile();
    tile[1].image =
ImageIO.read(getClass().getResourceAsStream("/tiles/wall.png"));
    tile[1].collsion = true;
```

```
        tile[2] = new Tile();
        tile[2].image =
ImageIO.read(getClass().getResourceAsStream("/tiles/water.png"));
        tile[2].collsion = true;
        ...
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

- *Loads tile images from the /tiles directory.*
- ***tile[0]** – Grass (walkable).*
- ***tile[1]** – Wall (collision enabled).*
- ***tile[2]** – Water (collision enabled).*
- ***tile[4]** – Tree (collision enabled).*
- *Each tile has an image, and some tiles (like walls and water) are marked with* ***collision = true*** *to block player movement.*

```
public void loadMap(String filePath) {
    try {
        InputStream is = getClass().getResourceAsStream(filePath);
        BufferedReader br = new BufferedReader(new InputStreamReader(is));

        int col = 0;
        int row = 0;
        while (col < gp.maxWorldCol && row < gp.maxWorldRow) {
            String line = br.readLine();
            while (col < gp.maxWorldCol) {
```

```
            String numbers[] = line.split(" ");
            int num = Integer.parseInt(numbers[col]);

            mapTileNum[col][row] = num;
            col++;
        }
        if (col == gp.maxWorldCol) {
            col = 0;
            row++;
        }
    }
    br.close();
  } catch (Exception e) {
  }
}
```

- *Reads map data from /maps/map01.txt.*
- *Each line represents a row of tiles, with numbers indicating tile types (like grass or wall).*
- ***split(" ")** – Breaks each line by spaces, storing tile numbers in **mapTileNum**.*
- *Each number corresponds to a tile in the tile[] array (e.g., 0 = grass, 1 = wall).*

```
public void draw(Graphics2D g2) {
  int worldCol = 0;
  int worldRow = 0;

  while (worldCol < gp.maxWorldCol && worldRow < gp.maxWorldRow) {
    int tileNum = mapTileNum[worldCol][worldRow];
```

```
        int worldX = worldCol * gp.titleSize;
        int worldY = worldRow * gp.titleSize;
        int screenX = worldX - gp.player.worldX + gp.player.screenX;
        int screenY = worldY - gp.player.worldY + gp.player.screenY;

        if (worldX + gp.titleSize > gp.player.worldX - gp.player.screenX &&
            worldX - gp.titleSize < gp.player.worldX + gp.player.screenX &&
            worldY + gp.titleSize > gp.player.worldY - gp.player.screenY &&
            worldY - gp.titleSize < gp.player.worldY + gp.player.screenY) {

            g2.drawImage(tile[tileNum].image, screenX, screenY, gp.titleSize,
gp.titleSize, null);
        }
        worldCol++;

        if (worldCol == gp.maxWorldCol) {
            worldCol = 0;
            worldRow++;
        }
    }
}
```

- ***Draws tiles*** *on the screen by looping through the map's columns and rows.*
- *Converts* ***world coordinates*** *to* ***screen coordinates*** *to account for player movement.*
- *Only tiles within the player's view (screen) are drawn for performance optimization.*
- ***g2.drawImage*** *– Draws each tile's image on the screen.*

CHAP 4: FINAL APP GAME

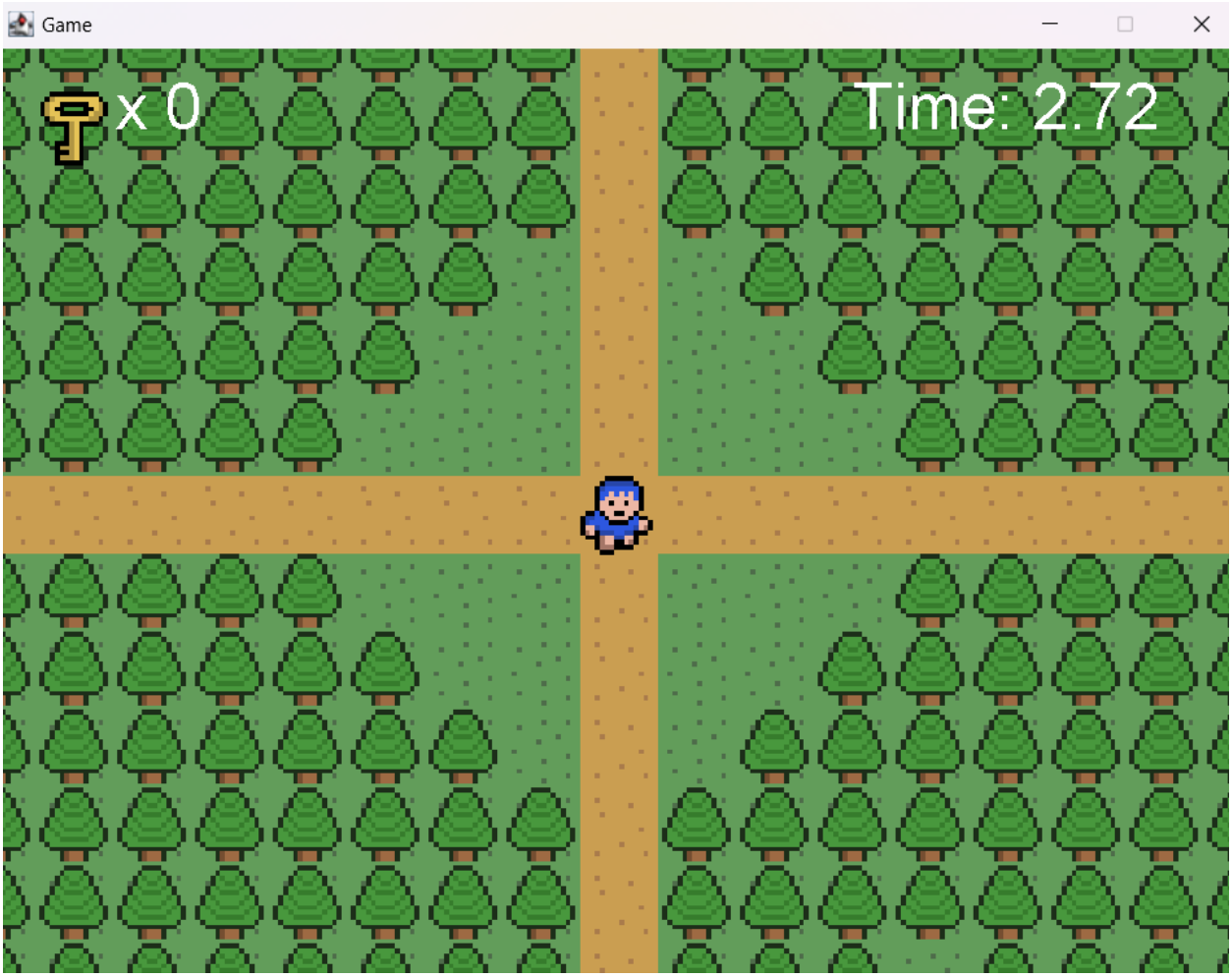
Source code (link github):

https://github.com/maiTienHung/Game

Instruction

1. **Begin the game:**

The gaming interface will be designed with simplicity in mind, allowing players to rapidly grasp and interact with it. It will have a clean and intuitive design that reduces distractions and allows users to focus on the games. A welcoming color design and easy navigation will create an inviting environment for players of all ages and skill levels. The goal is to make the experience as frictionless as possible while removing any barriers to entry. This technique will ensure that even inexperienced players may readily access and enjoy the game without undue difficulty.



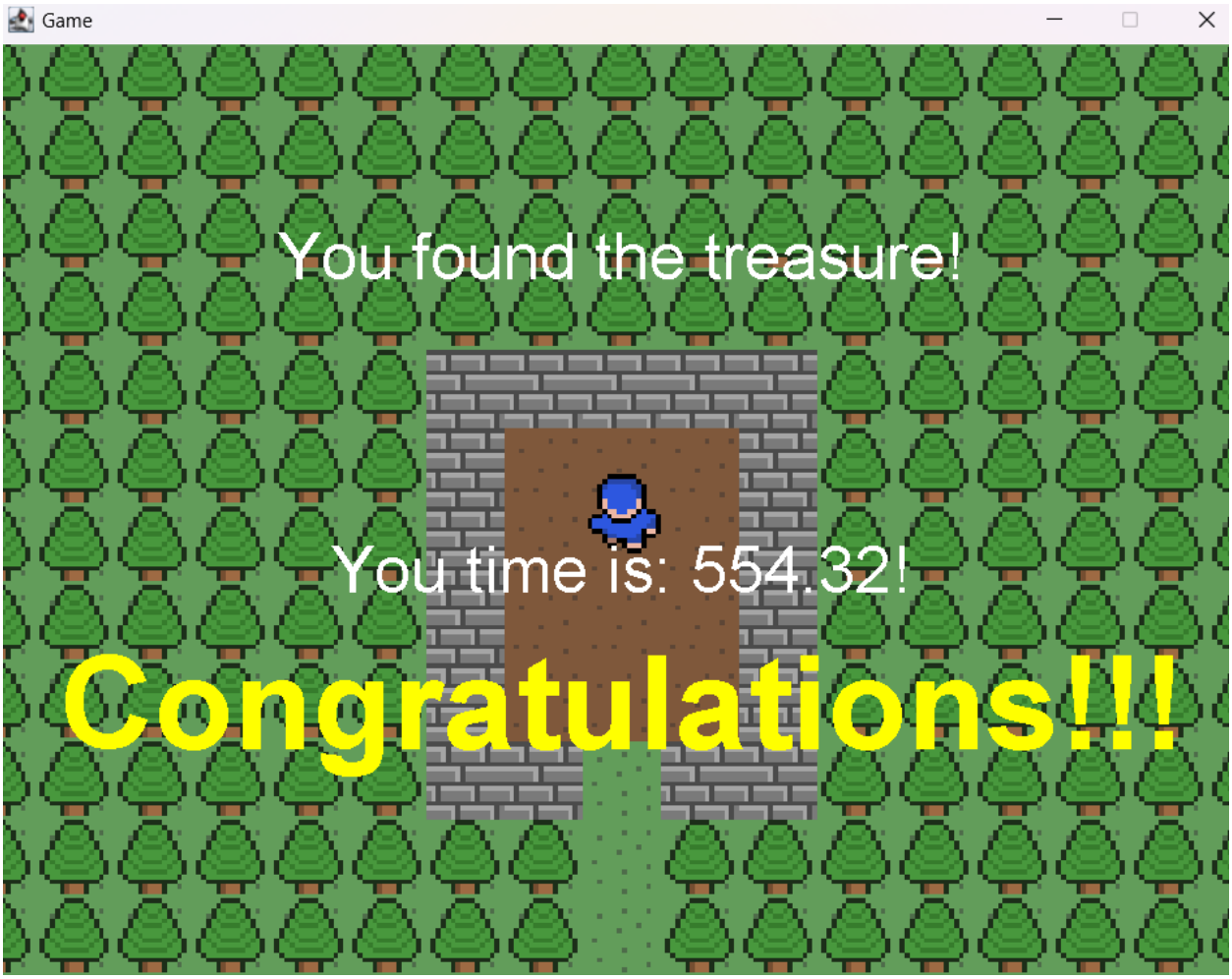
2. How to play

-Firstly, Use the W, A, S, and D keys to move the character in the indicated directions. Press W to move up, A to move left, S to move down, and D to move right. Guide the character to the target location by following the key prompts.

-Secondly, Once you pick up the key, the top left box will update to display the key number. A message will appear saying, "You have picked up the key." This lets you know the key has been successfully collected and added to your inventory.


Game
x 1
Time: 39.35

-Thirdly, start by locating the door near your initial position. Walk over the door to collect it, completing the first round. Afterward, return to the starting point to begin the next round. Repeat the process for two more rounds, finding and collecting the treasure each time. Completing all three rounds will mark the end of the level.



- After completing three rounds, the player who finishes first is declared the winner. The game ends once the first player successfully collects treasures and returns to the starting point.

CHAP 5: EXPERIENCE

After one week of working on this project, we can come to the following conclusion: Game development not only allows for practice, but it also increases the team's chemistry, resulting in improved teamwork.

In the end, our team concluded that teamwork is critical in the process of technological creation. And the information technology world is too large to learn alone in school.