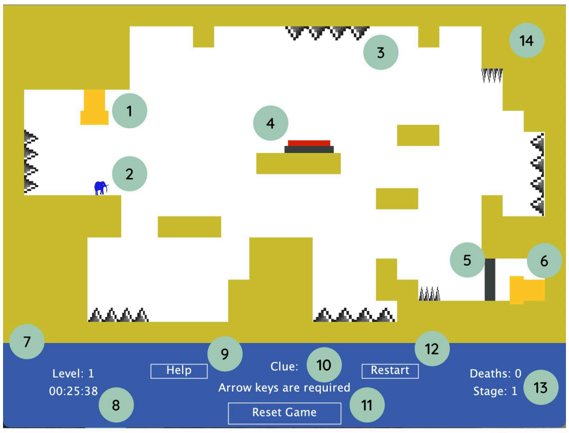
1. Introduction

[Gameplay Video](https://youtu.be/EnoGF8QM9Eo)(https://youtu.be/EnoGF8QM9Eo)
“This is the Only Level” is a game coming with a different approach for gaming. Instead of having several levels, this game contains one level which has diverse stages. Although all stages are placed on the same map, each stage has unique mechanics. Game has developed with Java and used the StdDraw library for GUI, which is a beginner level interface tool to use for education purposes from Pearson. To be more specific, the game contains OOP concepts and principles to allow 3rd party people to develop new features -more details are available in section 2.

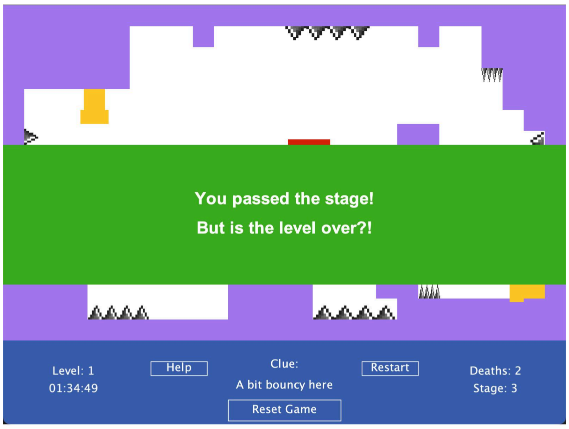
Player has the purpose of taking the elephant (illustrated in Figure 1, Object 2), which is dropped from startPipe (shown in Figure 1, Object 1), firstly to the button (illustrated in Figure 1, Object 4) to open the door(illustrated in Figure 1, Object 5), then continue to the exit pipe (illustrated in Figure 1, Object 6) by controlling it using arrow keys until they complete all stages. In addition, the map contains obstacles (illustrated in Figure 1, Object 14), or walls, which are not available to move on. Also, there are spikes (illustrated in Figure 1, Object 3) resetting the game as the elephant hits. If all stages are completed winner screen appears (illustrated in Figure 3)

Information bar (shown in Figure 1, Object 7) is developed to show several stats to players. It contains a time bar (shown in Figure 1, Object 8) which shows the time passed after the game begins, time passed when the transition page (shown in Figure 2) is visible is excluded. Help button (shown in Figure 1, Object 9) is used to trigger help visibility. Clue, or help, indicator (shown in Figure 1, Object 10) shows a slogan to the player initially, if the user triggers help, it transforms into a help text which directly informs the user about the stage. Reset Game (shown in Figure 1, Object 11) button is used to totally reset game parameters. Restart button (shown in Figure 1, Object 12) is used to restart the current stage. Scores (illustrated in Figure 1, Object 13) shows total death count from the game reset and current stage.



Level: 1
00:25:38
Clue:
Arrow keys are required
Deaths: 0
Stage: 1

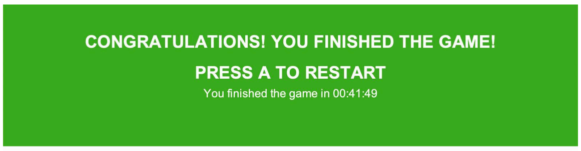
Figure 1



You passed the stage!
But is the level over?!

Level: 1
01:34:49
Clue:
A bit bouncy here
Deaths: 2
Stage: 3

Figure 2



CONGRATULATIONS! YOU FINISHED THE GAME!
PRESS A TO RESTART
You finished the game in 00:41:49

Developed by Ahmet Sait Denizli

Figure 3

2. Implementation

Game was developed within the frame of OOP and clean code principles. There are 4 classes which are responsible for different domains of the game. Although we have tried to distinguish domains of classes, some of them are related to each other. We have created a flow chart (shown Figure 6) to clarify the infrastructure. (More detailed version in last page)

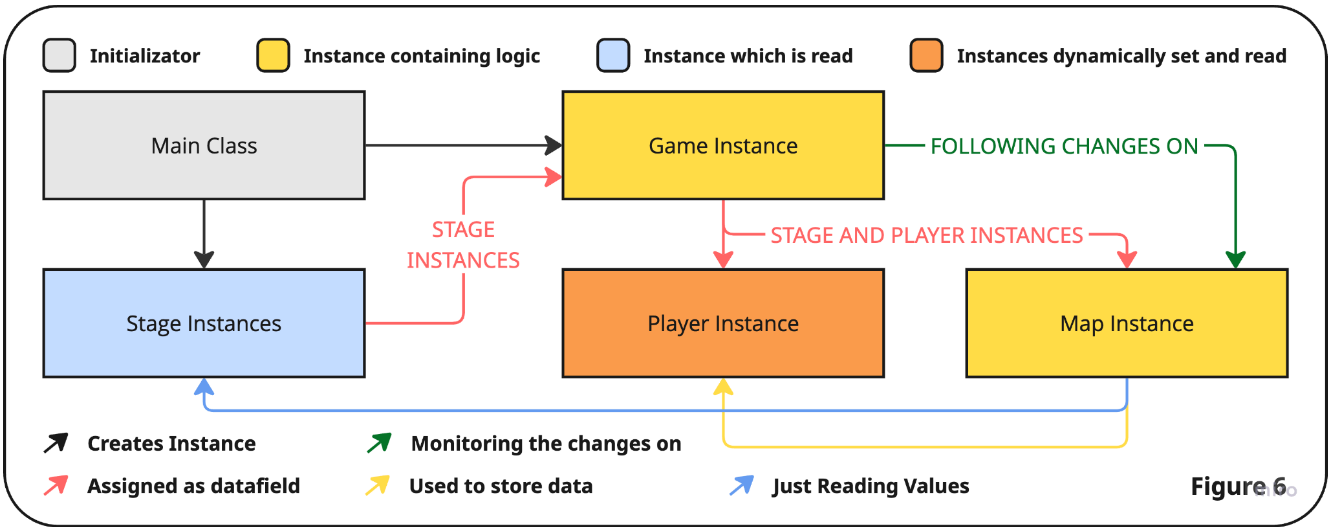


Figure 6

We have stored the coordinates of different GUI items in data fields in order to increase readability and correspond to magic number avoidance. Changeable, or which might be changeable as in the future, data fields are defined on top of the class block. However, since they may be altered, we have set them on constructors to increase flexibility of future development processes. In addition, constant data fields are defined and valued below the changeable data fields in the code. Please note that this documentation does not explain the methods which are accessors, mutators and responsible for basic operations.

2.1 Game Class

Game class is responsible for the main game cycle and initiating the GUI window and main elements such as information area.

2.1.1 Changeable Data Fields

Changeable, or which might be changeable as in the future, data fields are defined on top of the class block. However, since they may be altered, we have set them on constructors to increase flexibility of the development process.

```
private GAME_STATE gameState
```

This variable enables us to monitor game state, therefore, we can render the required components of the GUI and also handle the controller according to various keyboard mappings. GAME_STATE enum is also mentioned separately in Section 2.1.3

```
private int stageIndex
```

This variable is used to follow the instant stage.

```
private ArrayList<Stage> stages
```

We are storing stages in Game class to manage game state more efficiently. Moreover, we are taking this stage in the initialization state of Game class which allows 3rd party developers to add their custom stages to the game without changing implementation, unless it is complex.

```
private int deathNumber
```

This is used to store the total death number of the user during a game cycle in order to show that on the Information Area.

```
private int gameTime
```

This is used to monitor the total time elapsed during gameplay of one cycle.

```
private int resetTime
```

This enables us to store the initiation of the new game cycle, since we have to calculate time elapsed by considering the delays of the code execution.

```
private boolean resetGame
```

This is used to make an indication about game reset without increasing the complexity of code inside the game loop. After being true, it initiates the resetting procedure in the while loop which we have deeply mentioned in Section 2.1.2.

```
private boolean isHelperPressed
```

This is a gauge for GUI to show Clue or Help. It is triggered by the mouse handling which we will mention in Section 2.1.2, handleInput function.

2.1.1 Constant Data Fields

Constant data fields are defined and valued below the changeable data fields in the code.

```
private final int[] HELP_BUTTON_COORDS = new int[] {values…}
private final int[] RESTART_BUTTON_COORDS = new int[] {values…}
private final int[] RESET_GAME_BUTTON_COORDS = new int[] {values…}
private final int[] INFO_SECTION_COORDS = new int[] {values…}
```

These all represent the coordinates of various components which are fixed.

```
private final Color INFO_SECTION_COLOR = new Color(value…);
private final Font[] GAME_FONTS = new Font[] { values… };
```

These values declare the styling properties of components.

```
private static final int GAME_FPS = 60;
```

Shows the game FPS. Since it won’t be changed and used in most of the part of the game, it was declared as static.

2.1.2 Methods

```
public Game(ArrayList<Stage> stages)
```

Game class was developed with one constructor which is responsible for setting the changeable data to data fields. It also initiates GUI.

```
private void initializeGUI(ArrayList<Stage> stages)
```

GUI is initialized by this function. It also contains canvas sizes as local variables.

```
public void play()
```

Play function initiates the game cycle and contains the application life cycle.

```
while (true) {
     StdDraw.clear();
     if (this.resetGame)
this.resetGame(map);
     this.handleInput(map);
     switch (this.gameState) {
         case GAME_STATE.PLAYING -> { … }
         case GAME_STATE.WINNER -> { … }
     }
     StdDraw.show(1000 / GAME_FPS);
}
```

This cycle never ends, unless an error occurs or the user quits the game. It initiates main GUI function and controller handling with customization of game_state which has enabled us to track status in a dynamic and flexible manner.

```
private void renderInfoBar()
```

This function manages the information bar which is demonstrated on Section 1.

```
private boolean isButtonPressed(int[] buttonCoords)
```

This function is triggered when we have to handle a mouse click on button.

```
private void setNextStage()
```

This function manages the stage transitions, and also declares the winner state.

```
private void handleInput(Map map) {
    if (this.gameState == GAME_STATE.PLAYING) { … implementation}
    if (this.gameState == GAME_STATE.WINNER) { … implementation}
}
```

This function handles all mouse and keyboard inputs separately according to state.

```
private void drawBanner(args…), private void draw…Banner()
```

drawBanner(args…) method implements the component logic for every banner execution, it is triggered by draw…Banner() methods that have different specifications.

```
private void resetGame(Map map), private void restartStage(Map map)
```

These methods are triggered by play() which is monitoring the data fields.

2.1.3 Additionals

```
private enum GAME_STATE { INIT, PLAYING, WINNER }
```

INIT is used for initial game construction which enables feature developments to create an introduction page etc. PLAYING and WINNER is managed by the play() function.

2.2 Map Class

This class manages players movement, move handling and drawing the objects.

2.2.1 Constant Data Fields

```
final private Player player;                 |    final private Color PIPE_COLOR = ... ;
final private int[][] obstacles;             |    final private Color PIPE_COLOR = ... ;
final private int[] button;                  |    final private Color BUTTON_COLOR = ... ;
final private int[] buttonFloor;             |    final private Color FLOOR_COLOR = ... ;
final private int[][] startPipe;             |    final private int DOOR_ANIMATION_SPEED = ... ;
final private int[][] exitPipe;              |    final private String SPIKE_IMAGE = " ... ";
final private int[] door;                    |    final private String[] PLAYER_IMAGES = { ... };
final private int[][] spikes;                |
final private int[][] DOOR_INITIAL_COORDS;   |
```

Constant data fields are defined and valued below the changeable data fields in code.These all data fields are referring to constant values of map. Object data fields which are declared as int[] contain coordinates in format of (xInitial, yInitial, xEnd, yEnd). The other values are mainly related to animations, coloring and assets.

2.2.2 Changeable Data Field

```
private int buttonPressNum;          |    private boolean isStageCompleted;
private boolean isDoorOpen;          |    private boolean resetStage;
private boolean isButtonPressed;     |
```

Fields in the left column are used to track whether the user completed the current stage or not by Map class. However, the right column is tracked by Game class which manages game status

```
public Map(Stage stage, Player player)
```

Map class was initialized via one constructor which is used to set the changeable datafields.

```
public void draw()
public static void drawRectangleByCoordinates(args ... )
public static void drawPictureByCoordinates(args ... )
public static void drawTextByCoordinates(args ... )
public static void drawButton(args ... )
```

These methods are responsible for whole game rendering. Since some of them are just taking coordinates and executing rendering, we declared them as static to use in Game class

```
public void checkCollision(double nextX, double nextY, int[] obstacle)
```

This method checks the collisions by comparing overlap status of two coordinates. All collision shortcut methods are dependent on this method.

```
public void checkPlayerCollision(double nextX, double nextY)
```

This method checks the player collisions which may impact the movement of the player.

```
public void handlePlayerCollision(double nextX, double nextY)
```

This method handles the trigger system depending on the player's movement such as handling button click, exit pipe collision and deaths caused by spike collisions.

```
public void handleDoorAnimation()
```

This function used to add open-close effects to the door smoothly.

```
public void mapCycle() {
    this.updateAcceleration(player);
    this.updatePosition();
    this.handlePlayerCollision(player.getX(), player.getY());
    this.handleDoorAnimation();
}
```

Method which is responsible for physics and animations is called on every cycle of the lifecycle (which is also shown in Figure 7).

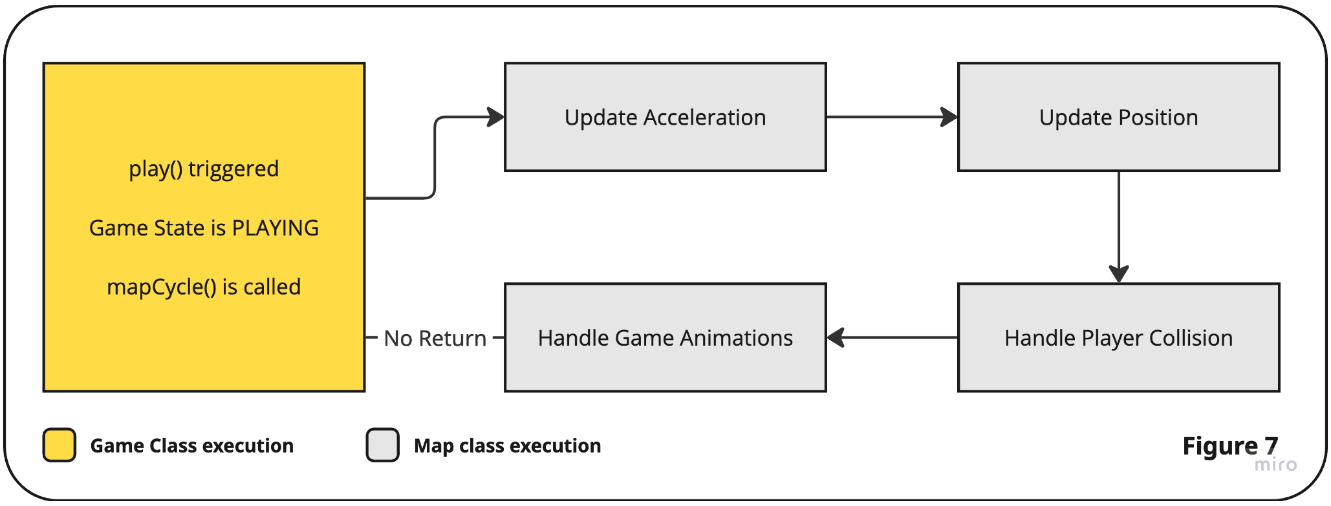


Figure 7

```
public void handleAcceleration(Player player)
```

This method updates vertical speed depending on gravity.

```
public void movePlayer(char direction) {
    // Some variables
    switch (direction) {
        case 'R' → {implementation…}
        case 'L' → {implementation…}
        case 'U' → {implementation…}
    }
}
```

This method contains a switch which canalizes the request to right movement. At the end of the code, it contains a player collision checker method to ensure that the player is able to move physically in the next frame.

```
public boolean changeStage()
```

This method is developed for Game class in order to monitor the stage status.

```
public boolean restartStage()
```

This method is used to reset all stage based parameters to default before stage initialization.

```
public void setStage(Stage stage)
```

It allows developers to switch stage and also calls restartStage to reset parameters.

```
private void updatePosition() {
    // other implementations
    double nextY = playerY + player.getVelocityY() / Game.GAME_FPS;
    if (checkPlayerCollision(playerX, nextY)) {
        player.setVelocityY(0);
        return;
    }
    player.setY(nextY);
}
```

This method implements the velocity handling model of the game. Algorithm (Figure 4) checks the player's next position initially. If the next position does not collide with any obstacle, we change the position. Otherwise, we set vertical velocity to zero by assuming it hits an obstacle and stops.

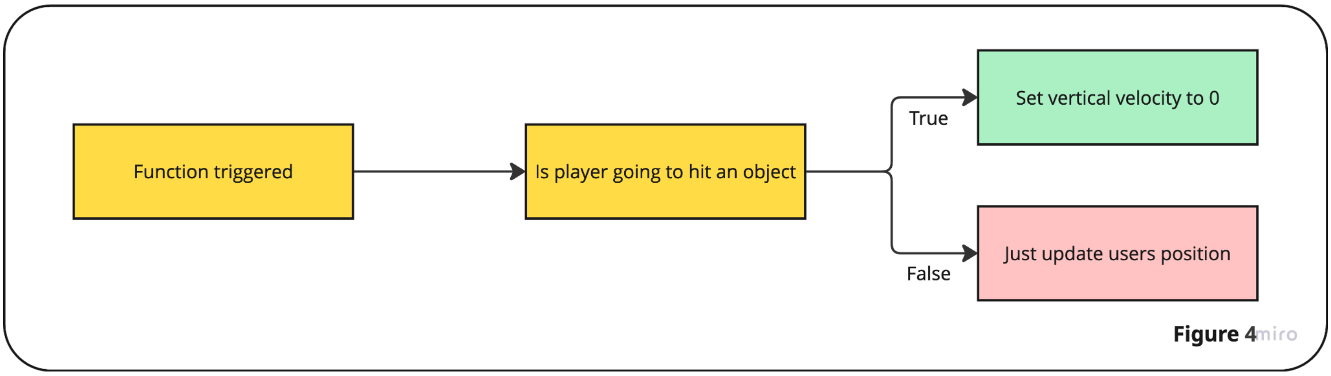

Function triggered
Is player going to hit an object
True
Set vertical velocity to 0
False
Just update users position
miro

Figure 4

2.3 Player Class

```
private double x;
private double y;
private double velocityY;
private char facing;
final private double width;
final private double height;
final private double[] initialCoords = {values … };

public Player()
public int[] getCoordinates()
public void resetPosition()
```

This class just used to store the player's object by a proper method. It contains only accessors and mutators except for resetPlayerCoordinates and specified getCoordinates functions. Unless the data field is defined as final, we set default data on the constructor which has no other responsibility.

2.4 Stage Class

```
final private int stageNumber;              |   final private int upCode;
final private double gravity;               |   final private double velocityX;
final private double velocityY;             |   final private int rightCode;
final private int leftCode;                 |   final private String clue;
final private String help;                  |   final private Color color;
final private int requiredNumberOfPress;
```

This class mainly stores each stage data as an object. All data fields except for color which is set randomly for every stage are set on a constructor which doesn't have any other aim.

```
public boolean validateDoor(int numberOfPress)
```

This function is used to transfer door validation logic to stage class which is responsible for stage logics. It is called from Map class. Initially just verifies number of presses but in ongoing developments it may be changed to an externally definable function in order to increase complexity of the game.

2.5 Additional Notes

2.5.1 State Flag Approach

Several handling mechanics are developed for this game. Some of them required ensuring execution of an action for once unless controller input is re-triggered. In other words, since we are using the StdDraw library to perceive controller inputs, we are not able to distinguish whether the input is triggered once or is the user holding the input. To overcome this problem, we implemented an algorithm which is called 'state-flag' approach which stores a boolean variable in instances to avoid handling many times even if the user hits the object once. To implement this follow following algorithm (also illustrated in Figure 5):

1. Create a recently handled variable (RCV) that will be changed when input is handled.
2. Execute functions if RCV is false, and immediately after execution set RCV to true.
3. Implement a callback mechanism for RCV in the handleInput() function, if the user releases the controller, set RCV to false. Therefore, subsequent controller inputs will be handled.

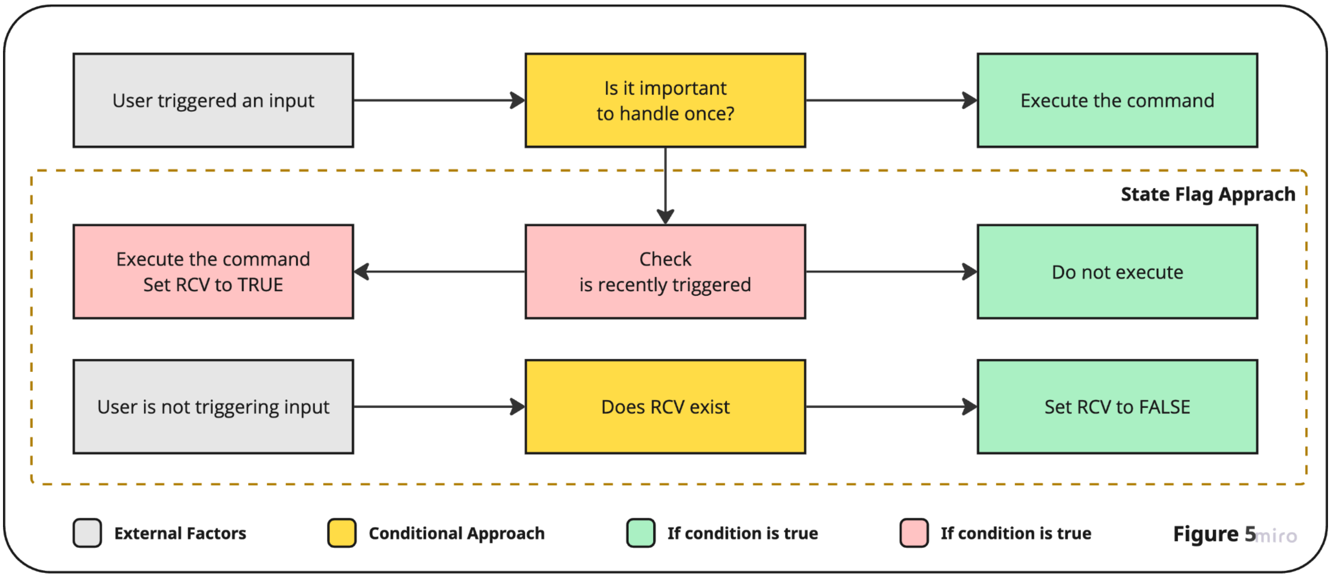


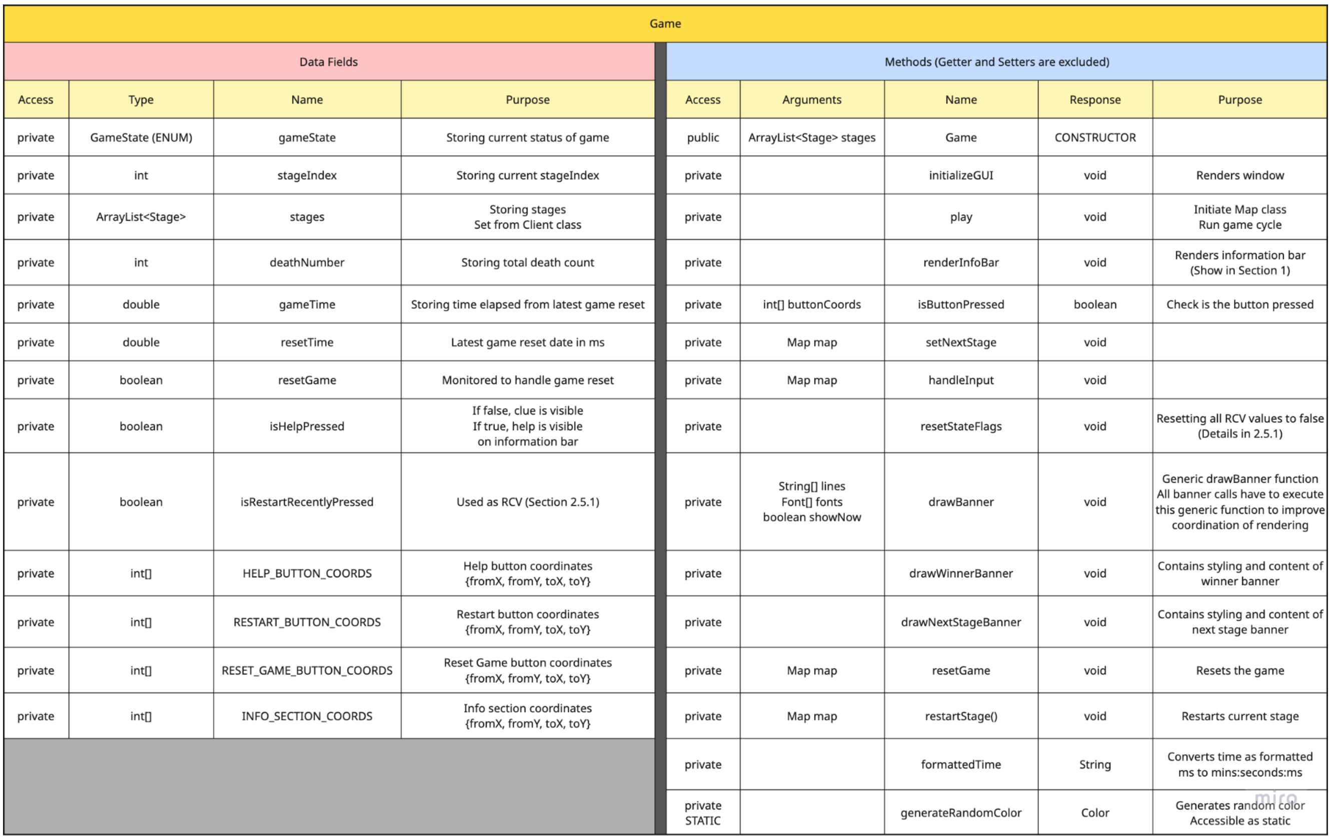
Figure 5

2.5.2 Game FPS

We are setting all physical data fields in stages as the movement amount that we want to handle in one second. Therefore, to achieve this, we are dividing the physical entries to FPS when we are setting them to related classes in order to fix the amount of movement to an integer value instead of a cycle. It solves the different gaming experience problems in different computers that have different specs.

For instance: nextY = playerY + stage.getVelocityY() / Game.GAME_FPS; to handle up movement.

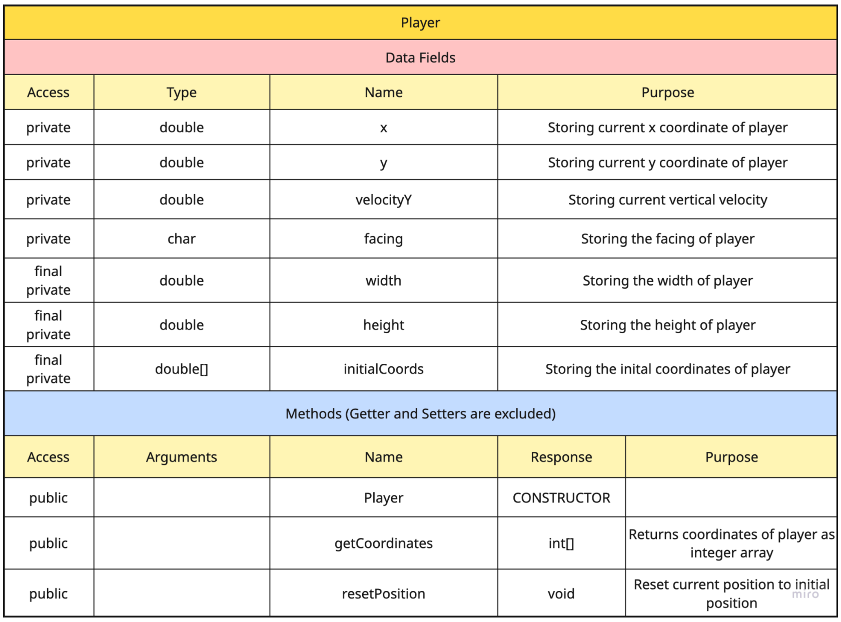
2.6 Quick Reference Tables



Game

Data Fields			
Access	Type	Name	Purpose
private	GameState (ENUM)	gameState	Storing current status of game
private	int	stageIndex	Storing current stageIndex
private	ArrayList<Stage>	stages	Storing stages Set from Client class
private	int	deathNumber	Storing total death count
private	double	gameTime	Storing time elapsed from latest game reset
private	double	resetTime	Latest game reset date in ms
private	boolean	resetGame	Monitored to handle game reset
private	boolean	isHelpPressed	If false, clue is visible If true, help is visible on information bar
private	boolean	isRestartRecentlyPressed	Used as RCV (Section 2.5.1)
private	int[]	HELP_BUTTON_COORDS	Help button coordinates {fromX, fromY, toX, toY}
private	int[]	RESTART_BUTTON_COORDS	Restart button coordinates {fromX, fromY, toX, toY}
private	int[]	RESET_GAME_BUTTON_COORDS	Reset Game button coordinates {fromX, fromY, toX, toY}
private	int[]	INFO_SECTION_COORDS	Info section coordinates {fromX, fromY, toX, toY}

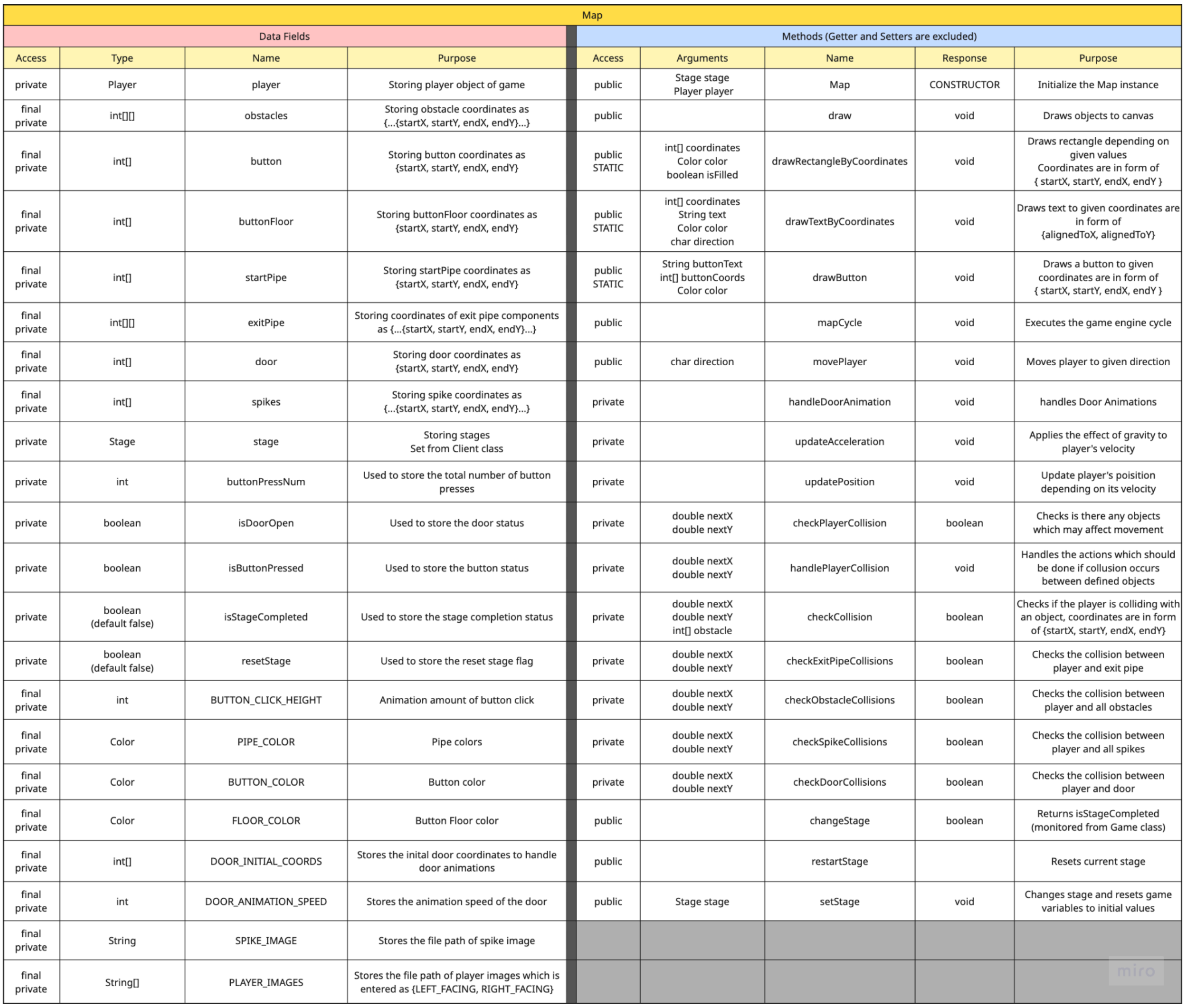
Methods (Getter and Setters are excluded)				
Access	Arguments	Name	Response	Purpose
public	ArrayList<Stage> stages	Game	CONSTRUCTOR	
private		initializeGUI	void	Renders window
private		play	void	Initiate Map class Run game cycle
private		renderInfoBar	void	Renders information bar (Show in Section 1)
private	int[] buttonCoords	isButtonPressed	boolean	Check is the button pressed
private	Map map	setNextStage	void	
private	Map map	handleInput	void	
private		resetStateFlags	void	Resetting all RCV values to false (Details in 2.5.1)
private	String[] lines Font[] fonts boolean showNow	drawBanner	void	Generic drawBanner function All banner calls have to execute this generic function to improve coordination of rendering
private		drawWinnerBanner	void	Contains styling and content of winner banner
private		drawNextStageBanner	void	Contains styling and content of next stage banner
private	Map map	resetGame	void	Resets the game
private	Map map	restartStage()	void	Restarts current stage
private		formattedTime	String	Converts time as formatted ms to mins:seconds:ms
private STATIC		generateRandomColor	Color	Generates random color Accessible as static



Player

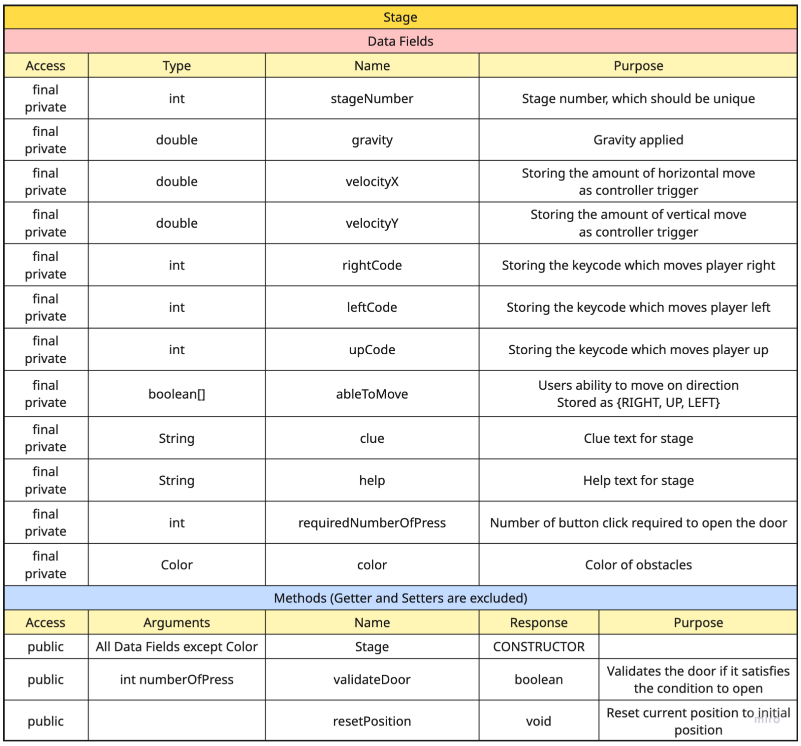
Data Fields			
Access	Type	Name	Purpose
private	double	x	Storing current x coordinate of player
private	double	y	Storing current y coordinate of player
private	double	velocityY	Storing current vertical velocity
private	char	facing	Storing the facing of player
final private	double	width	Storing the width of player
final private	double	height	Storing the height of player
final private	double[]	initialCoords	Storing the inital coordinates of player

Methods (Getter and Setters are excluded)				
Access	Arguments	Name	Response	Purpose
public		Player	CONSTRUCTOR	
public		getCoordinates	int[]	Returns coordinates of player as integer array
public		resetPosition	void	Reset current position to initial position



Map			
Data Fields			
Access	Type	Name	Purpose
private	Player	player	Storing player object of game
final private	int[][]	obstacles	Storing obstacle coordinates as {...{startX, startY, endX, endY}...}
final private	int[]	button	Storing button coordinates as {startX, startY, endX, endY}
final private	int[]	buttonFloor	Storing buttonFloor coordinates as {startX, startY, endX, endY}
final private	int[]	startPipe	Storing startPipe coordinates as {startX, startY, endX, endY}
final private	int[][]	exitPipe	Storing coordinates of exit pipe components as {...{startX, startY, endX, endY}...}
final private	int[]	door	Storing door coordinates as {startX, startY, endX, endY}
final private	int[]	spikes	Storing spike coordinates as {...{startX, startY, endX, endY}...}
private	Stage	stage	Storing stages Set from Client class
private	int	buttonPressNum	Used to store the total number of button presses
private	boolean	isDoorOpen	Used to store the door status
private	boolean	isButtonPressed	Used to store the button status
private	boolean (default false)	isStageCompleted	Used to store the stage completion status
private	boolean (default false)	resetStage	Used to store the reset stage flag
final private	int	BUTTON_CLICK_HEIGHT	Animation amount of button click
final private	Color	PIPE_COLOR	Pipe colors
final private	Color	BUTTON_COLOR	Button color
final private	Color	FLOOR_COLOR	Button Floor color
final private	int[]	DOOR_INITIAL_COORDS	Stores the inital door coordinates to handle door animations
final private	int	DOOR_ANIMATION_SPEED	Stores the animation speed of the door
final private	String	SPIKE_IMAGE	Stores the file path of spike image
final private	String[]	PLAYER_IMAGES	Stores the file path of player images which is entered as {LEFT_FACING, RIGHT_FACING}

Methods (Getter and Setters are excluded)				
Access	Arguments	Name	Response	Purpose
public	Stage stage Player player	Map	CONSTRUCTOR	Initialize the Map instance
public		draw	void	Draws objects to canvas
public STATIC	int[] coordinates Color color boolean isFilled	drawRectangleByCoordinates	void	Draws rectangle depending on given values Coordinates are in form of { startX, startY, endX, endY }
public STATIC	int[] coordinates String text Color color char direction	drawTextByCoordinates	void	Draws text to given coordinates are in form of {alignedToX, alignedToY}
public STATIC	String buttonText int[] buttonCoords Color color	drawButton	void	Draws a button to given coordinates are in form of { startX, startY, endX, endY }
public		mapCycle	void	Executes the game engine cycle
public	char direction	movePlayer	void	Moves player to given direction
private		handleDoorAnimation	void	handles Door Animations
private		updateAcceleration	void	Applies the effect of gravity to player's velocity
private		updatePosition	void	Update player's poisition depending on its velocity
private	double nextX double nextY	checkPlayerCollision	boolean	Checks is there any objects which may affect movement
private	double nextX double nextY	handlePlayerCollision	void	Handles the actions which should be done if collusion occurs between defined objects
private	double nextX double nextY int[] obstacle	checkCollision	boolean	Checks if the player is colliding with an object, coordinates are in form of {startX, startY, endX, endY}
private	double nextX double nextY	checkExitPipeCollisions	boolean	Checks the collision between player and exit pipe
private	double nextX double nextY	checkObstacleCollisions	boolean	Checks the collision between player and all obstacles
private	double nextX double nextY	checkSpikeCollisions	boolean	Checks the collision between player and all spikes
private	double nextX double nextY	checkDoorCollisions	boolean	Checks the collision between player and door
public		changeStage	boolean	Returns isStageCompleted (monitored from Game class)
public		restartStage		Resets current stage
public	Stage stage	setStage	void	Changes stage and resets game variables to initial values



Stage			
Data Fields			
Access	Type	Name	Purpose
final private	int	stageNumber	Stage number, which should be unique
final private	double	gravity	Gravity applied
final private	double	velocityX	Storing the amount of horizontal move as controller trigger
final private	double	velocityY	Storing the amount of vertical move as controller trigger
final private	int	rightCode	Storing the keycode which moves player right
final private	int	leftCode	Storing the keycode which moves player left
final private	int	upCode	Storing the keycode which moves player up
final private	boolean[]	ableToMove	Users ability to move on direction Stored as {RIGHT, UP, LEFT}
final private	String	clue	Clue text for stage
final private	String	help	Help text for stage
final private	int	requiredNumberOfPress	Number of button click required to open the door
final private	Color	color	Color of obstacles

Methods (Getter and Setters are excluded)				
Access	Arguments	Name	Response	Purpose
public	All Data Fields except Color	Stage	CONSTRUCTOR	
public	int numberOfPress	validateDoor	boolean	Validates the door if it satisfies the condition to open
public		resetPosition	void	Reset current position to initial position