ICT1009 Team 21

Flashy Road

MOHAMED ASIF S/O K A AMANULLAH	2000582
YIP HOU LIANG	2000721
MUHAMMAD NAZRI BIN SAPIAH	2000744
BOEY JING HENG TERENCE	2000772
AIN MUNIRAH BINTE MAKMOR	2002107
BEK ZI YING	2002175

Table Of Contents

Project Description

This project, “Flashy Road”, is a multiplayer game built with an object-oriented programming concept in mind. The team spent 4 weeks brainstorming, designing and creating the game. In this game, two players compete to overcome obstacles and transverse through game interactions to increase player scores. The player will win if the other player loses all his health. To simplify the project, “Flashy Road” consists of a game Graphic User Interface (GUI), game logic file, instructions screen, real-time scoreboard screen, 2-Dimensional map regeneration logic, characters classes, obstacles classes, and sound effect.

System Design and Features

A. <u>Main Screen</u>

When starting the game, the players are greeted with a start-up screen that shows the image of the background, the logo with the name of the game, the starting lives of the individual players, the option to view the game instructions or to start playing immediately. [START] will run the timer of the game and make the GUI invincible. [HOW TO PLAY] brings out a new panel that displays the game instructions. [LEADERBOARD] shows the scores of the different players. Both [HOW TO PLAY] and [LEADERBOARD] do not destroy the previous screen but instead create new windows for game continuation. There is a sound icon which allows the player to mute or unmute the background music.

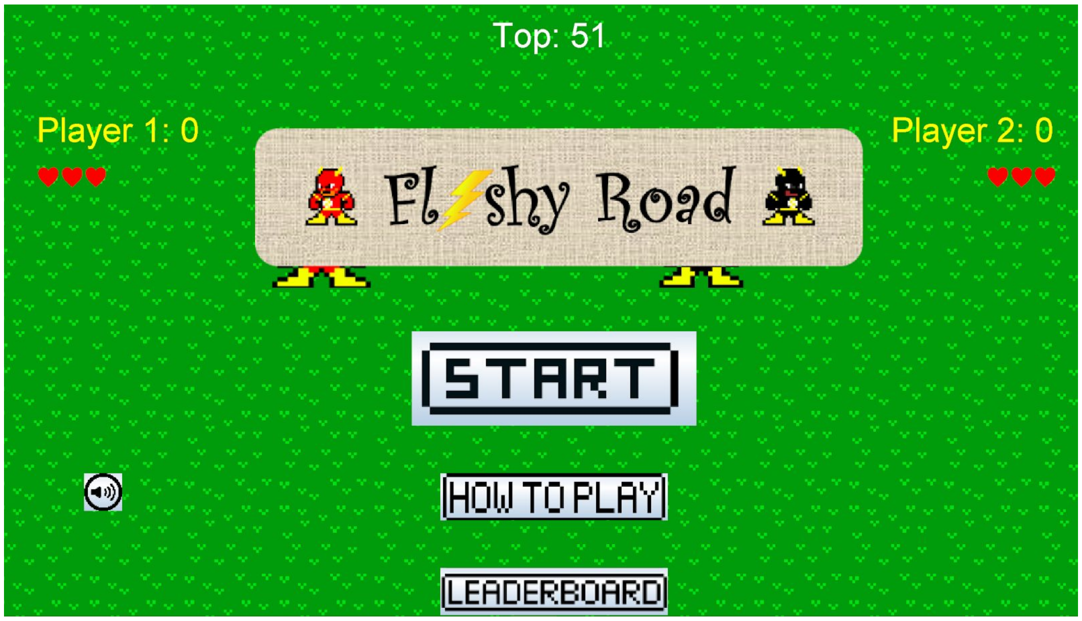
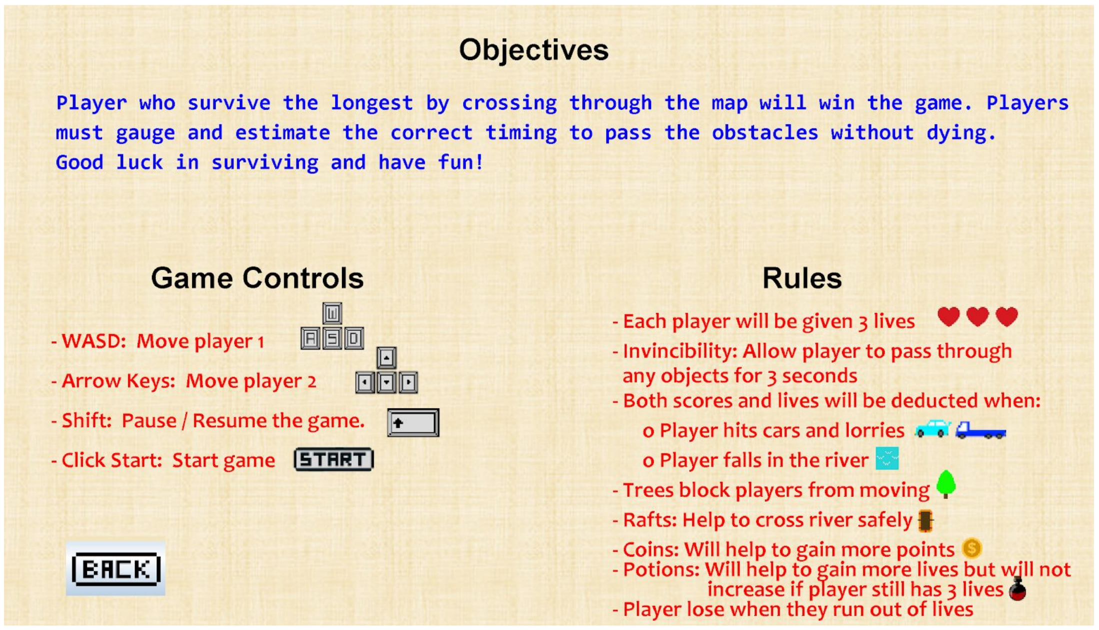


Figure 1: The start-up screen

B. Instructions

Clicking on the [HOW TO PLAY] will show the objectives, rules and controls of the game. Players can click the [BACK] button to return to the home screen page. The movement for the characters are registered from the input of the same keyboard. The characters move independently of one another. Player one can use the key W, A, S, D to move the sprite up, left, down and right while player two uses the 4 directional arrow keys. In Flashy Road, there are two types of item, Deadly Items and Friendly Items. The Deadly Items consist of vehicles, trees and rivers. These items perform different tasks. For example, the trees stop the character from advancing in a certain direction, if either the moving vehicles collide with the players or the players fall into the rivers, it will cause them to lose their scores and lives. Friendly Items are items that can help the players to win the game. It consists of rafts, potions and coins. The coins increase the player's score when collected while the potions are able to refill the player's lives. Rafts are on the rivers which help the players to cross rivers safely.



Objectives

Player who survive the longest by crossing through the map will win the game. Players must gauge and estimate the correct timing to pass the obstacles without dying. Good luck in surviving and have fun!

Game Controls

- WASD: Move player 1
- Arrow Keys: Move player 2
- Shift: Pause / Resume the game.
- Click Start: Start game

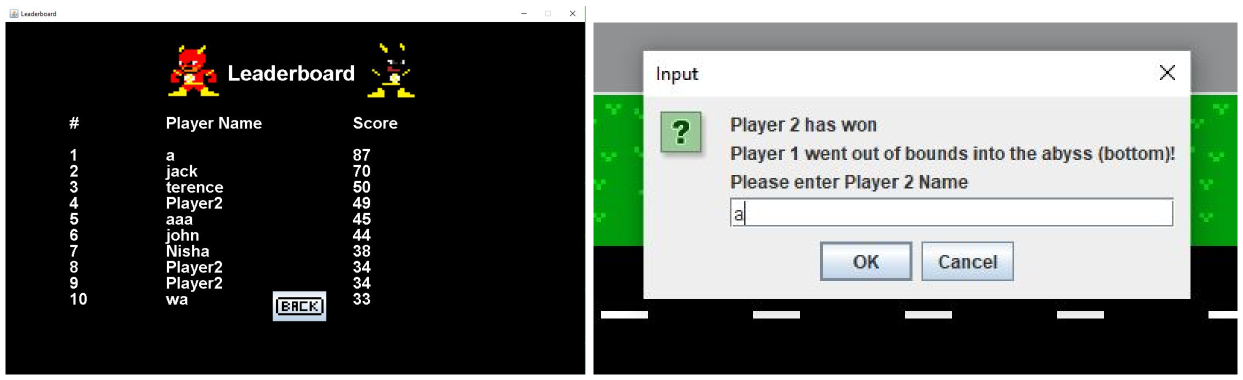
Rules

- Each player will be given 3 lives
- Invincibility: Allow player to pass through any objects for 3 seconds
- Both scores and lives will be deducted when:
  - o Player hits cars and lorries
  - o Player falls in the river
- Trees block players from moving
- Rafts: Help to cross river safely
- Coins: Will help to gain more points
- Potions: Will help to gain more lives but will not increase if player still has 3 lives
- Player lose when they run out of lives

Figure 2: Game instructions

C. Scoreboard

The player's scores are updated in real-time when running the game. When the game ends, the winner of that game round is able to input their name and the winner's score is being compared to all the previous high scores. If that high score is in one of the Top 10, then the game will update the scoreboard. The scoreboard will include the player's ranking, name and score. Whenever the game relaunches, it will read the file to display the Top 10 highest score recorded in the Leaderboard panel.



#	Player Name	Score
1	a	87
2	jack	70
3	terence	50
4	Player2	49
5	aaa	45
6	john	44
7	Nisha	38
8	Player2	34
9	Player2	34
10	wa	33

Input

Player 2 has won
Player 1 went out of bounds into the abyss (bottom)!
Please enter Player 2 Name

a

Figure 3: Leaderboard screen and pop-up box to enter winner's name.

D. Map Generation

The map was created by dividing the screen into multiple rows and columns of 13 by 12 using a 2D array. The sprites images are placed in each index of the array to form the map of the game that contains all the obstacles and characters. This process was done one row at a time to prevent cross-over of different types of surface such as having the river surface on the road or having obstruction such as a tree in the middle of the road. Each row also generates random surfaces and obstacles so that these elements do not remain fixed throughout the game.The map is randomly generated and everytime the user enters the game, the map will be different. Figure 4 depicts the overall design of the map at random.

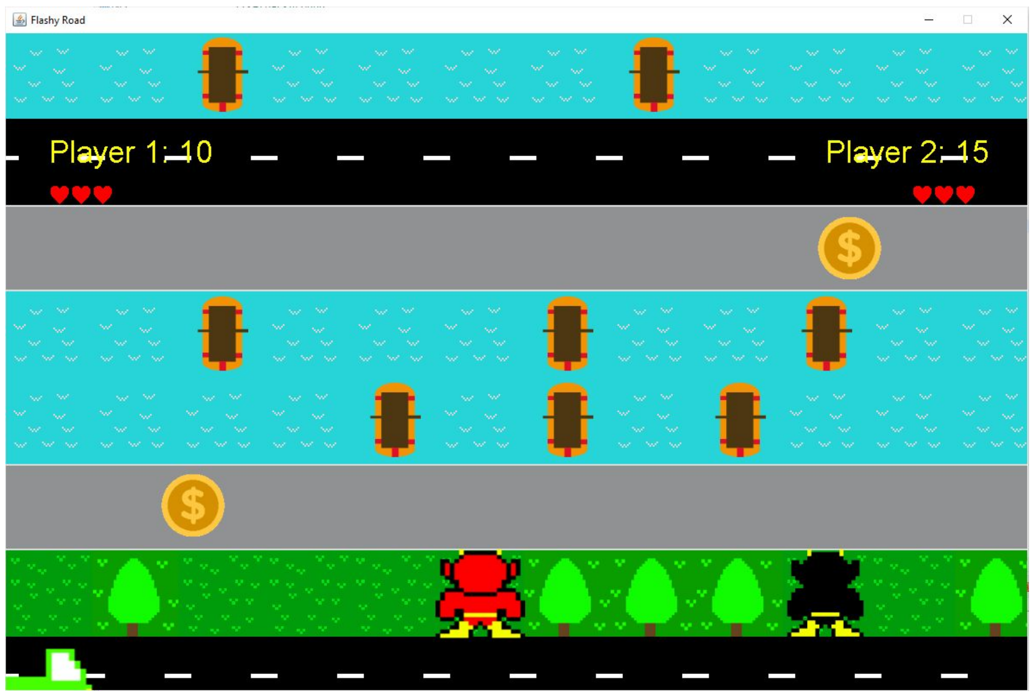


Figure 4: Obstacles on the map.

E. **Characters**

The characters were created by creating an object that contains four images for each character, for each direction the character is moving as shown in figure 5. Similar concept is applied to the vehicles however the difference is the direction of the vehicles remains the same when it moves from one end of the road to another. Therefore, constant change of direction throughout the motion is not required unless it is generated on another road whereby the direction of the vehicle moving is randomly generated to be either from left to right or vice versa. This means that for each type of vehicle, it requires 2 images, one facing the left and the other facing the right. Obstacles that do not require directional change such as the trees, raft and coins only require one image each. The same is applied to the sprites that are used at the background such as road, grass and water.

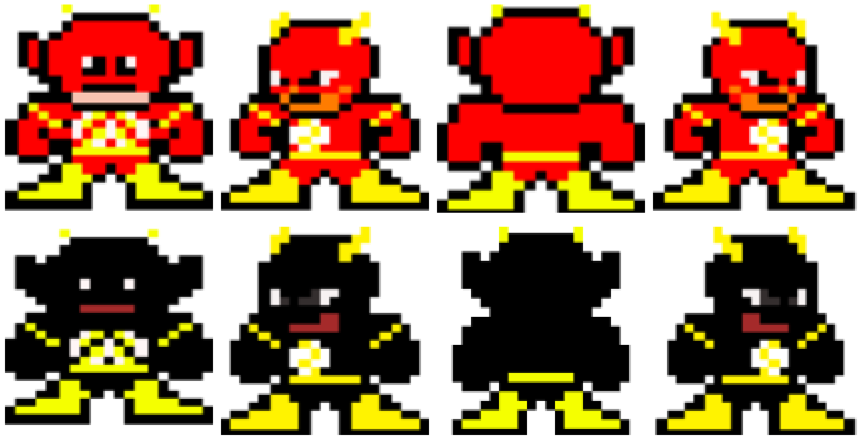


Figure 5: Sprite of the Flash Characters.

F. **Life and Collision System**

Collision with the vehicles, river or moving out of the screen reduces the player's lives individually. The number of hearts displayed at the top left and right of the game represent the remaining lives each player has. If any of the players loses all his life, the other player is declared the winner and a pop-up box will appear showing the causes of death, the highest score and the winner of the game as shown in figure 6 and 7.

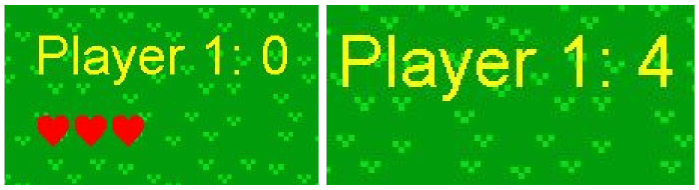
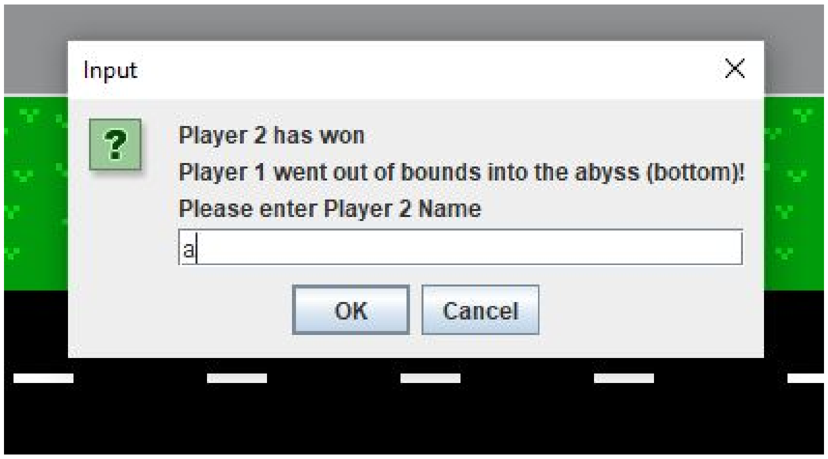


Figure 6: Health Bar of Players.


Input

Player 2 has won
Player 1 went out of bounds into the abyss (bottom)!
Please enter Player 2 Name

a

Figure 7: Pop-up JOptionPane to Display Death Message and Winner

G. <u>Sound Effects</u>

The Sound effects are called from wav files. The audio system library is used for this case, Figure 8. Separately the main class, FlashyRoad will call the background music. The sound calling methods are independent for different classes and therefore we used static to global the sound class. This allows us to call the clip, which stores in the sound from other classes to pause the sound.

Additional sounds are called concurrently when the characters collide into the objects. There is also an interface implemented to add different player sounds emitting when collided to give them an individual persona. There is also a mute button available for players to turn off the music as shown in Figure 9.

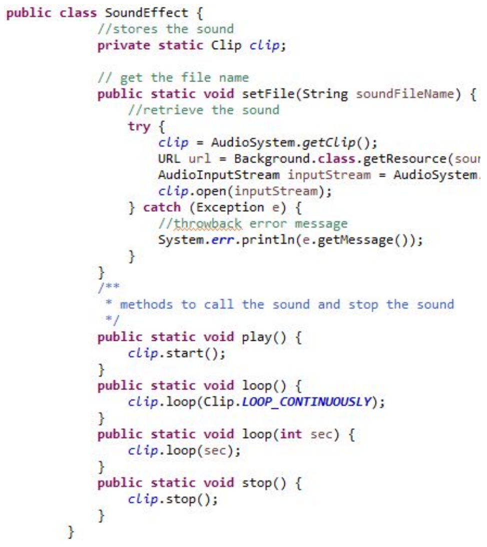

```
public class SoundEffect {
        //stores the sound
        private static Clip clip;

        // get the file name
        public static void setFile(String soundFileName) {
            //retrieve the sound
            try {
                clip = AudioSystem.getClip();
                URL url = Background.class.getResource(sour
                AudioInputStream inputStream = AudioSystem.
                clip.open(inputStream);
            } catch (Exception e) {
                //throwback error message
                System.err.println(e.getMessage());
            }
        }
        /**
         * methods to call the sound and stop the sound
         */
        public static void play() {
            clip.start();
        }
        public static void loop() {
            clip.loop(Clip.LOOP_CONTINUOUSLY);
        }
        public static void loop(int sec) {
            clip.loop(sec);
        }
        public static void stop() {
            clip.stop();
        }
    }
```

Figure 8: SoundEffect class

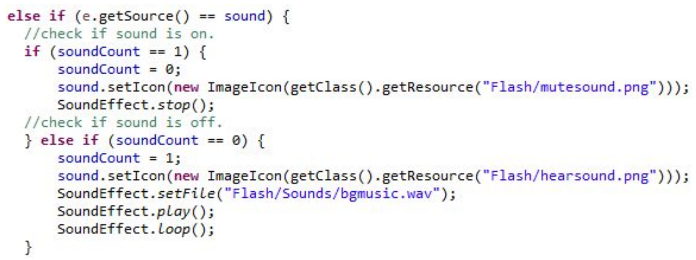

```
else if (e.getSource() == sound) {
  //check if sound is on.
  if (soundCount == 1) {
      soundCount = 0;
      sound.setIcon(new ImageIcon(getClass().getResource("Flash/mutesound.png")));
      SoundEffect.stop();
  //check if sound is off.
  } else if (soundCount == 0) {
      soundCount = 1;
      sound.setIcon(new ImageIcon(getClass().getResource("Flash/hearsound.png")));
      SoundEffect.setFile("Flash/Sounds/bgmusic.wav");
      SoundEffect.play();
      SoundEffect.loop();
  }
```

Figure 9: Declared Sound Method

UML Diagram

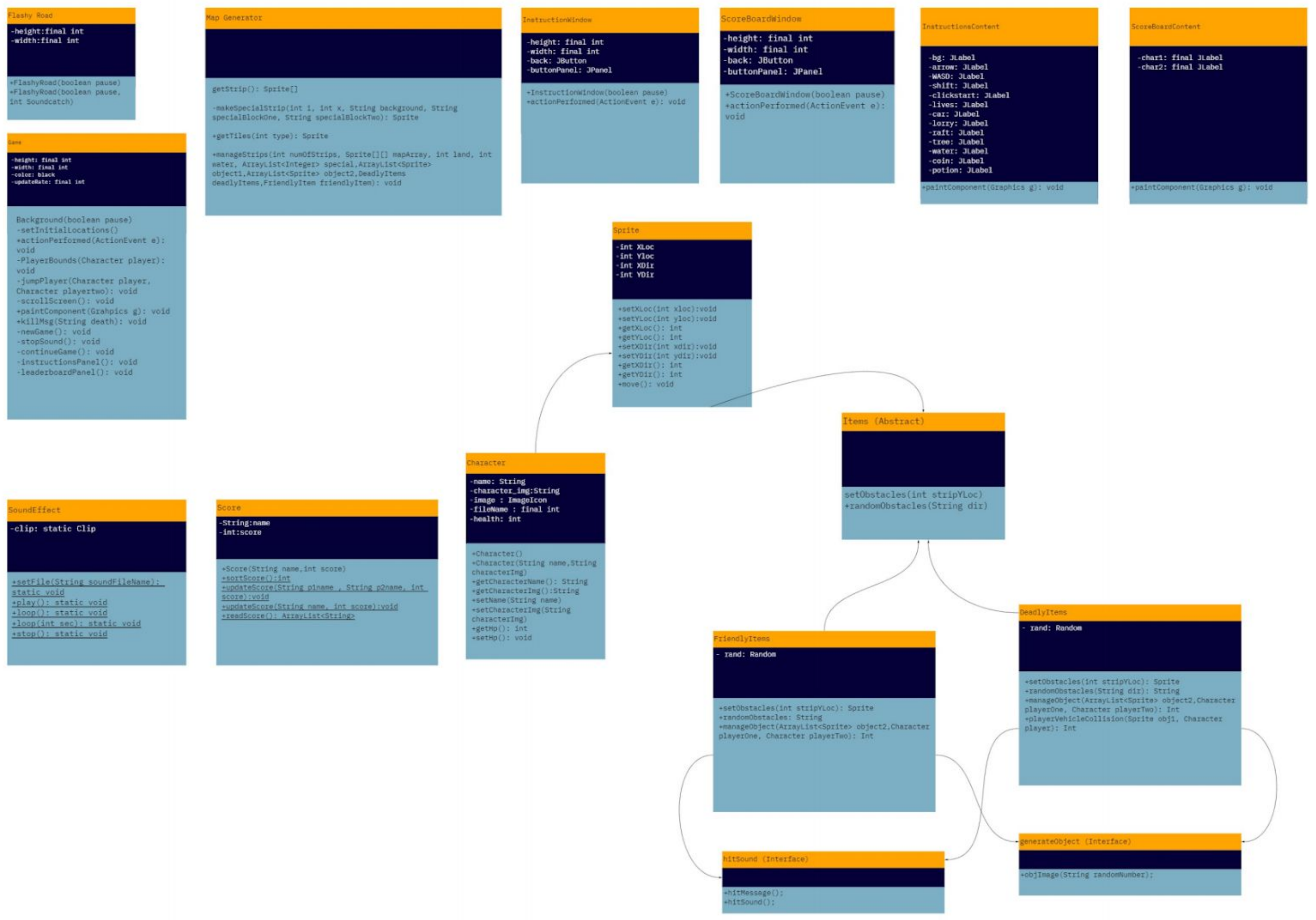
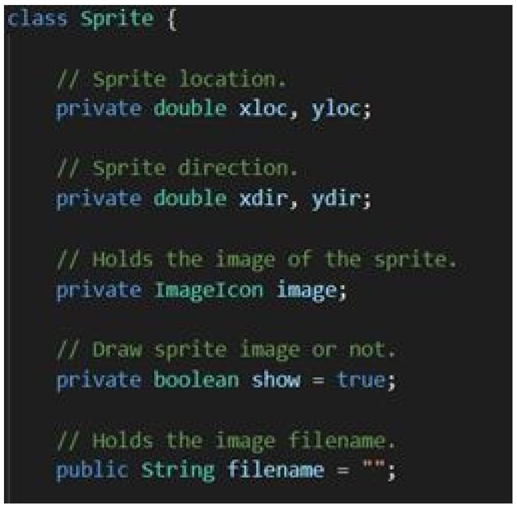

Flashy Road
-height:final int
-width:final int
+FlashyRoad(boolean pause)
+FlashyRoad(boolean pause, int Soundcatch)
Map Generator
getStrip(): Sprite[]
-makeSpecialStrip(int i, int x, String background, String specialBlockOne, String specialBlockTwo): Sprite
+getTiles(int type): Sprite
+manageStrips(int numOfStrips, Sprite[][] mapArray, int land, int water, ArrayList<Integer> special,ArrayList<Sprite> object1,ArrayList<Sprite> object2,DeadlyItems deadlyItems,FriendlyItem friendlyItem): void
InstructionWindow
-height: final int
-width: final int
-back: JButton
-buttonPanel: JPanel
+InstructionWindow(boolean pause)
+actionPerformed(ActionEvent e): void
ScoreBoardWindow
-height: final int
-width: final int
-back: JButton
-buttonPanel: JPanel
+ScoreBoardWindow(boolean pause)
+actionPerformed(ActionEvent e): void
InstructionsContent
-bg: JLabel
-arrow: JLabel
-WASD: JLabel
-shift: JLabel
-clickstart: JLabel
-lives: JLabel
-car: JLabel
-lorry: JLabel
-raft: JLabel
-tree: JLabel
-water: JLabel
-coin: JLabel
-potion: JLabel
+paintComponent(Graphics g): void
ScoreBoardContent
-char1: final JLabel
-char2: final JLabel
+paintComponent(Graphics g): void
Game
-height: final int
-width: final int
-color: black
-updateRate: final int
Background(boolean pause)
-setInitialLocations()
+actionPerformed(ActionEvent e): void
-PlayerBounds(Character player): void
-jumpPlayer(Character player, Character playertwo): void
-scrollScreen(): void
+paintComponent(Grahpics g): void
+killMsg(String death): void
-newGame(): void
-stopSound(): void
-continueGame(): void
-instructionsPanel(): void
-leaderboardPanel(): void
Sprite
-int XLoc
-int Yloc
-int XDir
-int YDir
+setXLoc(int xloc):void
+setYLoc(int yloc):void
+getXLoc(): int
+getYLoc(): int
+setXDir(int xdir):void
+setYDir(int ydir):void
+getXDir(): int
+getYDir(): int
+move(): void
Items (Abstract)
setObstacles(int stripYLoc)
+randomObstacles(String dir)
Character
-name: String
-character_img:String
-image : ImageIcon
-fileName : final int
-health: int
+Character()
+Character(String name,String characterImg)
+getCharacterName(): String
+getCharacterImg():String
+setName(String name)
+setCharacterImg(String characterImg)
+getHp(): int
+setHp(): void
SoundEffect
-clip: static Clip
+setFile(String soundFileName): static void
+play(): static void
+loop(): static void
+loop(int sec): static void
+stop(): static void
Score
-String:name
-int:score
+Score(String name,int score)
+sortScore():int
+updateScore(String p1name , String p2name, int score):void
+updateScore(String name, int score):void
+readScore(): ArrayList<String>
FriendlyItems
- rand: Random
+setObstacles(int stripYLoc): Sprite
+randomObstacles: String
+manageObject(ArrayList<Sprite> object2,Character playerOne, Character playerTwo): Int
DeadlyItems
- rand: Random
+setObstacles(int stripYLoc): Sprite
+randomObstacles(String dir): String
+manageObject(ArrayList<Sprite> object2,Character playerOne, Character playerTwo): Int
+playerVehicleCollision(Sprite obj1, Character player): Int
hitSound (Interface)
+hitMessage();
+hitSound();
generateObject (Interface)
+objImage(String randomNumber);

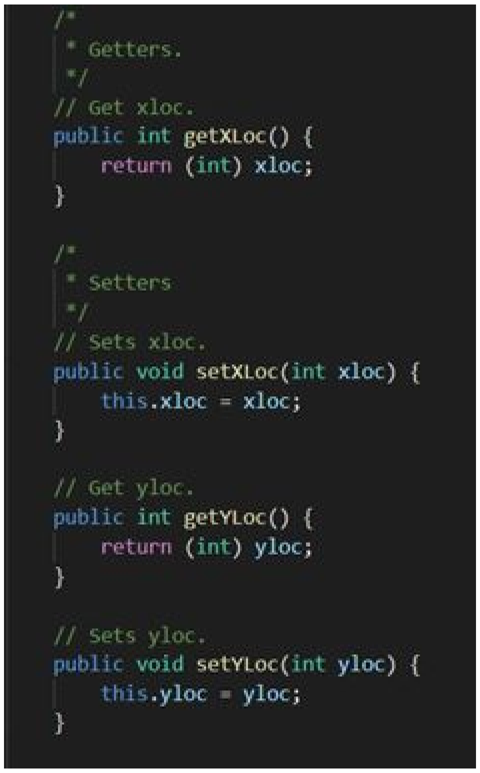
Figure 10: Flashy Road UML diagram

Object Oriented Principles

A. Encapsulation



```
class Sprite {

    // Sprite location.
    private double xloc, yloc;

    // Sprite direction.
    private double xdir, ydir;

    // Holds the image of the sprite.
    private ImageIcon image;

    // Draw sprite image or not.
    private boolean show = true;

    // Holds the image filename.
    public String filename = "";
```



```
/*
 * Getters.
 */
// Get xloc.
public int getXLoc() {
    return (int) xloc;
}

/*
 * Setters
 */
// Sets xloc.
public void setXLoc(int xloc) {
    this.xloc = xloc;
}

// Get yloc.
public int getYLoc() {
    return (int) yloc;
}

// Sets yloc.
public void setYLoc(int yloc) {
    this.yloc = yloc;
}
```

Figure 11: Code Snippet of Encapsulation

The snippet above is a small example of how encapsulation has been implemented on the project. Creating various getters and setters to encapsulate private variables. Purpose of encapsulating was to simplify maintenance and reduce human errors while coding.

B. Inheritance

We have employed inheritance through the Multi-Level Inheritance structure. As displayed in the UML diagram, the Sprite class will be the super class, while the Character, Items, FriendlyItems, and lastly DeadlyItems will be the subclasses.

The purpose of structuring the codes this way was to allow code simplicity and reusability. With this implementation, everything generated on the map will be inheriting methods from the Sprite class. The sprite class contains the methods to get, set and collision methods. Which is inherited by every object in the game, allowing the ease of code reusability.

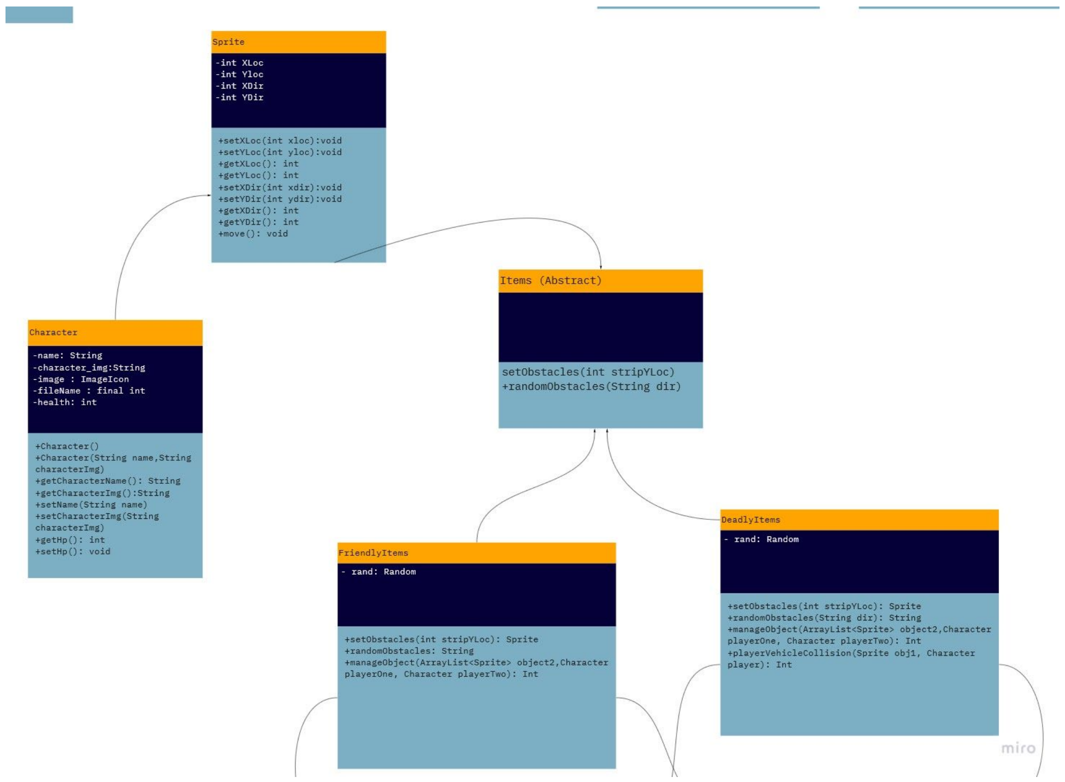
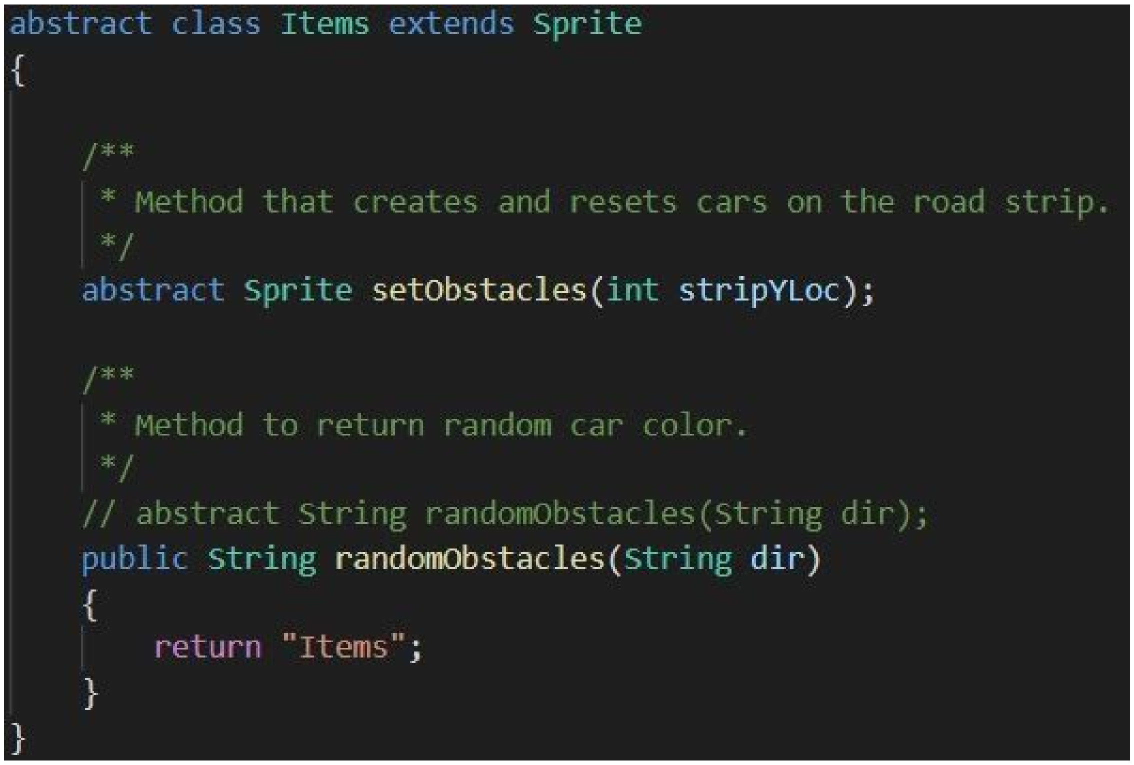

Sprite
-int XLoc
-int Yloc
-int XDir
-int YDir
+setXLoc(int xloc):void
+setYLoc(int yloc):void
+getXLoc(): int
+getYLoc(): int
+setXDir(int xdir):void
+setYDir(int ydir):void
+getXDir(): int
+getYDir(): int
+move(): void
Character
-name: String
-character_img:String
-image : ImageIcon
-fileName : final int
-health: int
+Character()
+Character(String name,String characterImg)
+getCharacterName(): String
+getCharacterImg():String
+setName(String name)
+setCharacterImg(String characterImg)
+getHp(): int
+setHp(): void
Items (Abstract)
setObstacles(int stripYLoc)
+randomObstacles(String dir)
FriendlyItems
- rand: Random
+setObstacles(int stripYLoc): Sprite
+randomObstacles: String
+manageObject(ArrayList<Sprite> object2,Character playerOne, Character playerTwo): Int
DeadlyItems
- rand: Random
+setObstacles(int stripYLoc): Sprite
+randomObstacles(String dir): String
+manageObject(ArrayList<Sprite> object2,Character playerOne, Character playerTwo): Int
+playerVehicleCollision(Sprite obj1, Character player): Int
miro

Figure 12: A snippet of the sprite class and the inheritance from UML diagram

C. Abstract Classes

The Item class has been set as an abstract class. There are two reasons for it. First, there is an abstract method within the class. Second, there should not be any initialisation of Item object anywhere in the game. Only the subclasses are meant to be initialised.

Within the Item class, setObstacles has also been set as an abstract. This method has been set as abstract to force its subclasses to have this method implemented. The setObstacles is the key function which returns the sprite object to be placed on the map, we implement the method this way to prevent this method from getting missed out. With this, we are also implementing Polymorphism (Overriding), which will be explained further below.



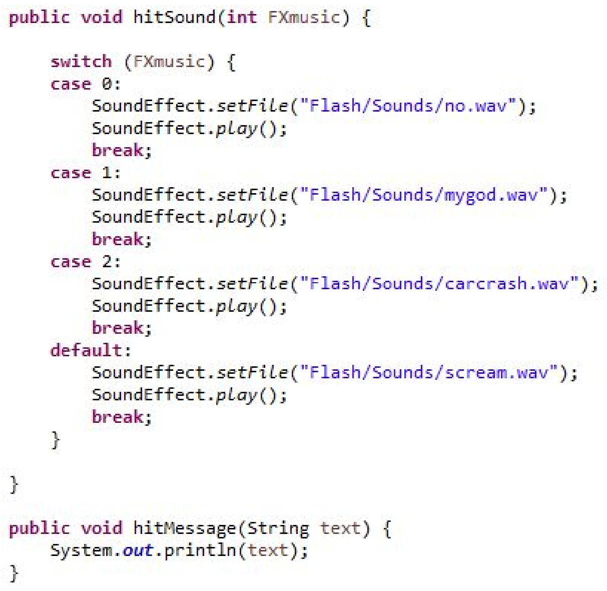
```
abstract class Items extends Sprite
{

    /**
     * Method that creates and resets cars on the road strip.
     */
    abstract Sprite setObstacles(int stripYLoc);

    /**
     * Method to return random car color.
     */
    // abstract String randomObstacles(String dir);
    public String randomObstacles(String dir)
    {
        return "Items";
    }
}
```

Figure 13: Code snippet of the abstract class Items

D. <u>Interfaces</u>

a. Hit Effects

This interface contains two methods, hitMessage and hitSound. hitMessage displays the object collision with the players. The methods are used in FriendlyItems and DeadlyItems classes which have different objects and therefore will print different message and collision sounds.



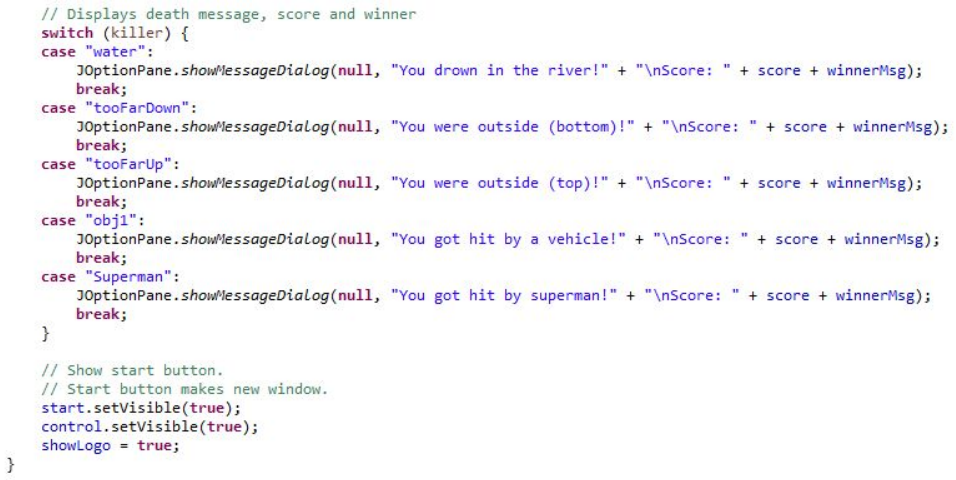
```
public void hitSound(int FXmusic) {

    switch (FXmusic) {
    case 0:
        SoundEffect.setFile("Flash/Sounds/no.wav");
        SoundEffect.play();
        break;
    case 1:
        SoundEffect.setFile("Flash/Sounds/mygod.wav");
        SoundEffect.play();
        break;
    case 2:
        SoundEffect.setFile("Flash/Sounds/carcrash.wav");
        SoundEffect.play();
        break;
    default:
        SoundEffect.setFile("Flash/Sounds/scream.wav");
        SoundEffect.play();
        break;
    }

}

public void hitMessage(String text) {
    System.out.println(text);
}
```

Figure 14: Switch statement for different death scenarios

When the last life of the player is reduced due to collision with obstacle or straying away from the frame, the following *kill message* will be displayed followed by the score of the winning player and the winner message. The winner message simply displays which player is the winner for that match. If players have at least 1 life after straying or collison, the players will respawn. Each at a different starting location upon respawn.

Each obstacle has its own *kill message*

- Water = “You drown in the river!”
- Stray too far up = “You were outside(top)!”
- Stray too far down = “You were outside(bottom)!”
- Cars or trucks = “You got hit by a vehicle!”

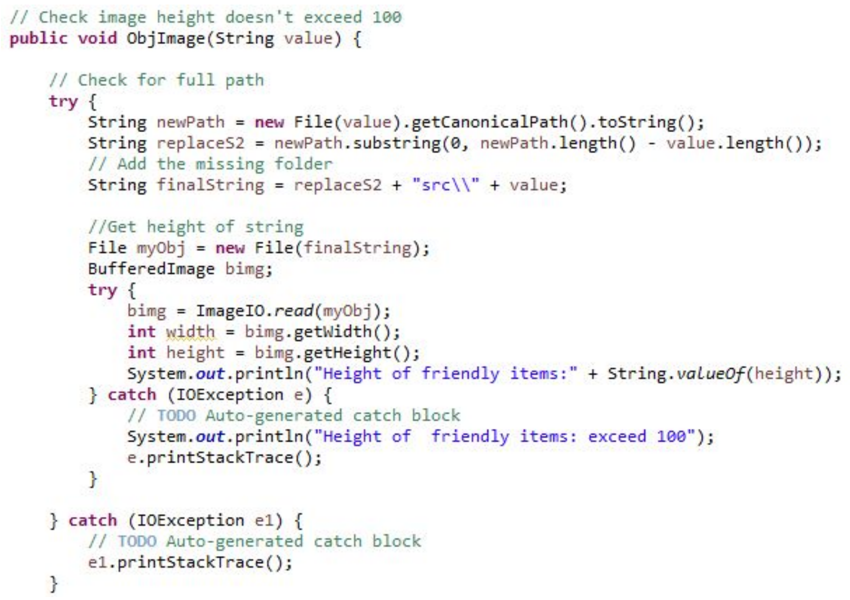


```
    // Displays death message, score and winner
    switch (killer) {
    case "water":
        JOptionPane.showMessageDialog(null, "You drown in the river!" + "\nScore: " + score + winnerMsg);
        break;
    case "tooFarDown":
        JOptionPane.showMessageDialog(null, "You were outside (bottom)!" + "\nScore: " + score + winnerMsg);
        break;
    case "tooFarUp":
        JOptionPane.showMessageDialog(null, "You were outside (top)!" + "\nScore: " + score + winnerMsg);
        break;
    case "obj1":
        JOptionPane.showMessageDialog(null, "You got hit by a vehicle!" + "\nScore: " + score + winnerMsg);
        break;
    case "Superman":
        JOptionPane.showMessageDialog(null, "You got hit by superman!" + "\nScore: " + score + winnerMsg);
        break;
    }

    // Show start button.
    // Start button makes new window.
    start.setVisible(true);
    control.setVisible(true);
    showLogo = true;
}
```

Figure 15: Switch statement for different death scenarios

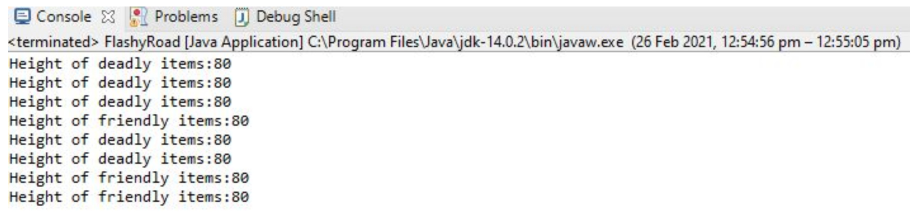
b. Object Size Checker

This interface allows the programmer to further use for future improvements. It retrieves the obstacle size after they are rendered on the screen. As the grid is only 100 pixels by 100, the object size cannot exceed 100 or else they will have a logic error, for example a bigger hitbox and smaller space for characters to navigate around.



```
// Check image height doesn't exceed 100
public void ObjImage(String value) {

    // Check for full path
    try {
        String newPath = new File(value).getCanonicalPath().toString();
        String replaceS2 = newPath.substring(0, newPath.length() - value.length());
        // Add the missing folder
        String finalString = replaceS2 + "src\\" + value;

        //Get height of string
        File myObj = new File(finalString);
        BufferedImage bimg;
        try {
            bimg = ImageIO.read(myObj);
            int width = bimg.getWidth();
            int height = bimg.getHeight();
            System.out.println("Height of friendly items:" + String.valueOf(height));
        } catch (IOException e) {
            // TODO Auto-generated catch block
            System.out.println("Height of  friendly items: exceed 100");
            e.printStackTrace();
        }

    } catch (IOException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
```

Figure 16: Check the Height of the Objects/Obstacles



```
Console ☒  Problems  Debug Shell
<terminated> FlashyRoad [Java Application] C:\Program Files\Java\jdk-14.0.2\bin\javaw.exe (26 Feb 2021, 12:54:56 pm – 12:55:05 pm)
Height of deadly items:80
Height of deadly items:80
Height of deadly items:80
Height of friendly items:80
Height of deadly items:80
Height of deadly items:80
Height of friendly items:80
Height of friendly items:80
```

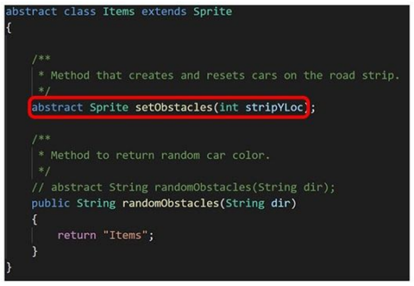
Figure 17: Output of Object Size Checker

E. <u>Polymorphism</u>

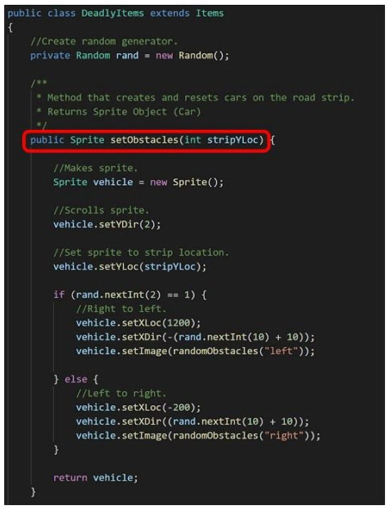
Items class is structured to be the super class. While the FriendlyItems and DeadlyItems classes are subclasses. Among the 3 classes, there are methods with similar names, but with our Polymorphism implementation, it allows objects of different types to be accessed through the same java file.

a. Overriding

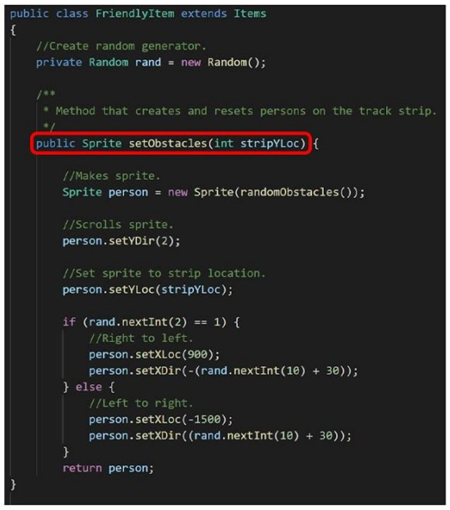
As mentioned above, we have achieved overriding with the method called setObstacles. Where the method is forced to be implemented in the subclasses with the same signature. This allows the subclasses to override the method and return the type of objects they require.



```java
abstract class Items extends Sprite
{

    /**
     * Method that creates and resets cars on the road strip.
     */
    abstract Sprite setObstacles(int stripYLoc);

    /**
     * Method to return random car color.
     */
    // abstract String randomObstacles(String dir);
    public String randomObstacles(String dir)
    {
        return "Items";
    }
}
```



```java
public class DeadlyItems extends Items
{
    //Create random generator.
    private Random rand = new Random();

    /**
     * Method that creates and resets cars on the road strip.
     * Returns Sprite Object (Car)
     */
    public Sprite setObstacles(int stripYLoc) {

        //Makes sprite.
        Sprite vehicle = new Sprite();

        //Scrolls sprite.
        vehicle.setYDir(2);

        //Set sprite to strip location.
        vehicle.setYLoc(stripYLoc);

        if (rand.nextInt(2) == 1) {
            //Right to left.
            vehicle.setXLoc(1200);
            vehicle.setXDir(-(rand.nextInt(10) + 10));
            vehicle.setImage(randomObstacles("left"));

        } else {
            //Left to right.
            vehicle.setXLoc(-200);
            vehicle.setXDir((rand.nextInt(10) + 10));
            vehicle.setImage(randomObstacles("right"));
        }

        return vehicle;
    }
```

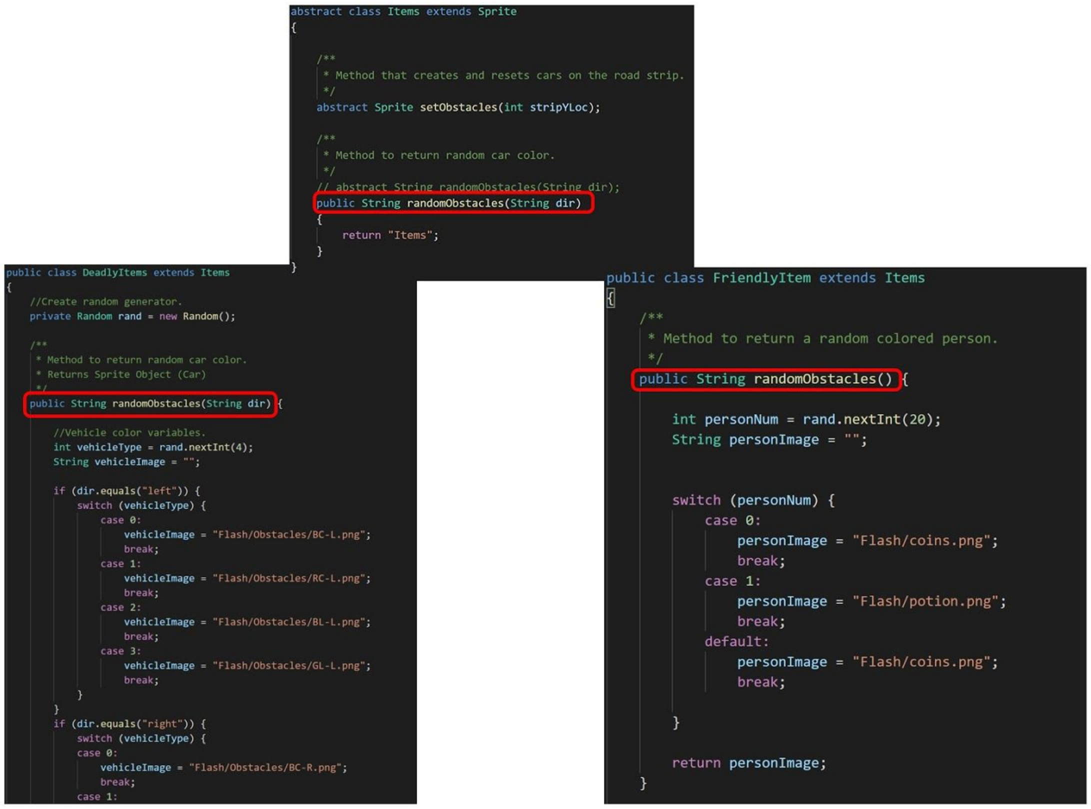


```java
public class FriendlyItem extends Items
{
    //Create random generator.
    private Random rand = new Random();

    /**
     * Method that creates and resets persons on the track strip.
     */
    public Sprite setObstacles(int stripYLoc) {

        //Makes sprite.
        Sprite person = new Sprite(randomObstacles());

        //Scrolls sprite.
        person.setYDir(2);

        //Set sprite to strip location.
        person.setYLoc(stripYLoc);

        if (rand.nextInt(2) == 1) {
            //Right to left.
            person.setXLoc(900);
            person.setXDir(-(rand.nextInt(10) + 30));
        } else {
            //Left to right.
            person.setXLoc(-1500);
            person.setXDir((rand.nextInt(10) + 30));
        }
        return person;
}
```

Figure 18: Code Snippet for Overriding Implementation

b. Overloading

Overloading is being implemented with randomObstacles method. This method can be found in Items, FriendlyItems and DeadlyItems classes. As randomObstacles randomises the objects that the setObstacles require, we have set the method arguments to be different and overload the function. With this, each subclass can choose the arguments they want to send into randomObstacles.



```java
abstract class Items extends Sprite
{

    /**
     * Method that creates and resets cars on the road strip.
     */
    abstract Sprite setObstacles(int stripYLoc);

    /**
     * Method to return random car color.
     */
    // abstract String randomObstacles(String dir);
    public String randomObstacles(String dir)
    {
        return "Items";
    }
}
```

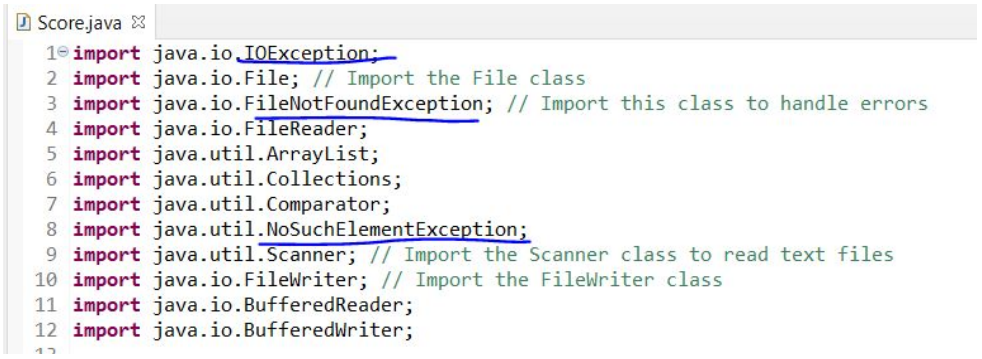
```java
public class DeadlyItems extends Items
{
    //Create random generator.
    private Random rand = new Random();

    /**
     * Method to return random car color.
     * Returns Sprite Object (Car)
     */
    public String randomObstacles(String dir) {

        //Vehicle color variables.
        int vehicleType = rand.nextInt(4);
        String vehicleImage = "";

        if (dir.equals("left")) {
            switch (vehicleType) {
                case 0:
                    vehicleImage = "Flash/Obstacles/BC-L.png";
                    break;
                case 1:
                    vehicleImage = "Flash/Obstacles/RC-L.png";
                    break;
                case 2:
                    vehicleImage = "Flash/Obstacles/BL-L.png";
                    break;
                case 3:
                    vehicleImage = "Flash/Obstacles/GL-L.png";
                    break;
            }
        }
        if (dir.equals("right")) {
            switch (vehicleType) {
            case 0:
                vehicleImage = "Flash/Obstacles/BC-R.png";
                break;
            case 1:
```

```java
public class FriendlyItem extends Items
{
    /**
     * Method to return a random colored person.
     */
    public String randomObstacles() {

        int personNum = rand.nextInt(20);
        String personImage = "";


        switch (personNum) {
            case 0:
                personImage = "Flash/coins.png";
                break;
            case 1:
                personImage = "Flash/potion.png";
                break;
            default:
                personImage = "Flash/coins.png";
                break;

        }

        return personImage;
    }
```

Figure 19: Code Snippet for Overloading Implementation

F. Error Handling

In our project FlashyRoad, we utilized exception handling through try catch blocks. In our Score class, we handle exceptions that are related to reading and writing external files, such as text file (.txt) and datastream file(.dat). Some examples used include IOException ,FileNotFoundException as well as NoSuchElementException.

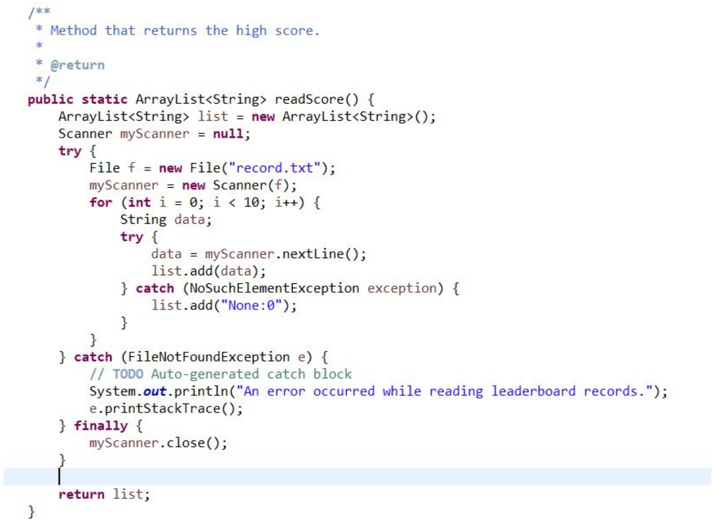


Score.java

```java
import java.io.IOException;
import java.io.File; // Import the File class
import java.io.FileNotFoundException; // Import this class to handle errors
import java.io.FileReader;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.NoSuchElementException;
import java.util.Scanner; // Import the Scanner class to read text files
import java.io.FileWriter; // Import the FileWriter class
import java.io.BufferedReader;
import java.io.BufferedWriter;
```

Figure 20: Imports of the Exception classes

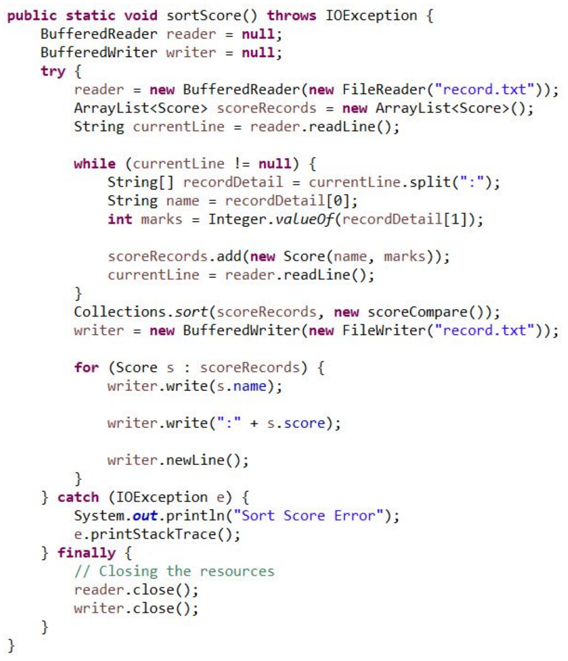
For this method which returns an ArrayList<String> that is used to generate the leaderboard array. The try catch block will catch any errors and throw a FileNotFoundException when the file is not found. The NoSuchElementexception in this case is used to generate the list of players names and scores as we will only display the top 10 scores, if the file does not contain 10 scores, the loop will hit the exception and replace the empty line with a void score.



```java
/**
 * Method that returns the high score.
 *
 * @return
 */
public static ArrayList<String> readScore() {
    ArrayList<String> list = new ArrayList<String>();
    Scanner myScanner = null;
    try {
        File f = new File("record.txt");
        myScanner = new Scanner(f);
        for (int i = 0; i < 10; i++) {
            String data;
            try {
                data = myScanner.nextLine();
                list.add(data);
            } catch (NoSuchElementException exception) {
                list.add("None:0");
            }
        }
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        System.out.println("An error occurred while reading leaderboard records.");
        e.printStackTrace();
    } finally {
        myScanner.close();
    }

    return list;
}
```

Figure 21: Code snippet of FileNotFoundException and NoSuchElementException

Similarly, we catch the IOException whenever we are handling reading/writing files. This will allow us to not break the flow of the game whenever there are issues with accessing the external files and error handling will allow us to efficiently debug and identify which area to look into to rectify it.



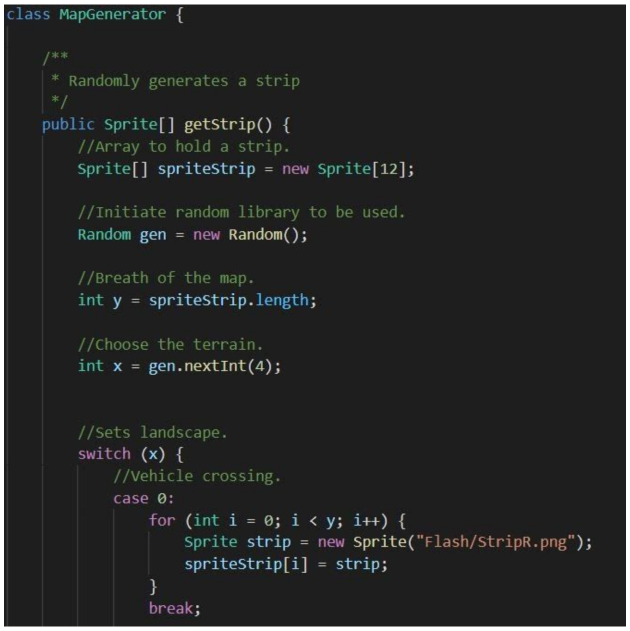
```java
public static void sortScore() throws IOException {
    BufferedReader reader = null;
    BufferedWriter writer = null;
    try {
        reader = new BufferedReader(new FileReader("record.txt"));
        ArrayList<Score> scoreRecords = new ArrayList<Score>();
        String currentLine = reader.readLine();

        while (currentLine != null) {
            String[] recordDetail = currentLine.split(":");
            String name = recordDetail[0];
            int marks = Integer.valueOf(recordDetail[1]);

            scoreRecords.add(new Score(name, marks));
            currentLine = reader.readLine();
        }
        Collections.sort(scoreRecords, new scoreCompare());
        writer = new BufferedWriter(new FileWriter("record.txt"));

        for (Score s : scoreRecords) {
            writer.write(s.name);

            writer.write(":" + s.score);

            writer.newLine();
        }
    } catch (IOException e) {
        System.out.println("Sort Score Error");
        e.printStackTrace();
    } finally {
        // Closing the resources
        reader.close();
        writer.close();
    }
}
```

Figure 22: Code snippet of IOException

Unique Features

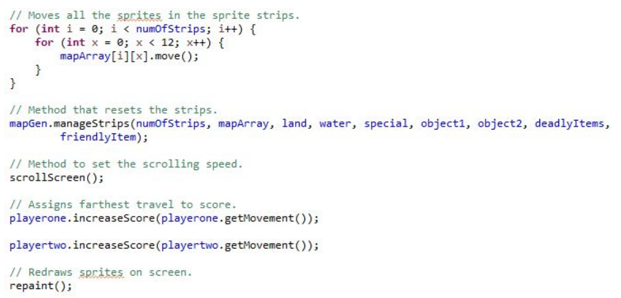
A. Map Generation and Rendering

The map is rendered by calling different images into each grid which is 100 by 100 for the height and width in pixels. The map is generated stripe by stripe, where 1 stripe consists of 12 tiles. The stripes will be stored into an array, which gets rendered by paintComponent function from Jframe. The game size is 1200x800, and each sprite object in size of tile 100x100 and the mapArray[13][12] to handle the collisions and also the displacement of the sprites due to the scrolling effect of the game. The logic of the terrains are predetermined, such that trees will appear on grass, rafts will appear in water only. Figure 24 shows a snippet of how the game randomly generates the terrains and figure 25 demonstrates how the map is scrolled vertically according to an inbuilt timer.



```java
class MapGenerator {

    /**
     * Randomly generates a strip
     */
    public Sprite[] getStrip() {
        //Array to hold a strip.
        Sprite[] spriteStrip = new Sprite[12];

        //Initiate random library to be used.
        Random gen = new Random();

        //Breath of the map.
        int y = spriteStrip.length;

        //Choose the terrain.
        int x = gen.nextInt(4);


        //Sets landscape.
        switch (x) {
            //Vehicle crossing.
            case 0:
                for (int i = 0; i < y; i++) {
                    Sprite strip = new Sprite("Flash/StripR.png");
                    spriteStrip[i] = strip;
                }
                break;
```

Figure 24: Code snippet of Map Rendering



```java
// Moves all the sprites in the sprite strips.
for (int i = 0; i < numOfStrips; i++) {
    for (int x = 0; x < 12; x++) {
        mapArray[i][x].move();
    }
}

// Method that resets the strips.
mapGen.manageStrips(numOfStrips, mapArray, land, water, special, object1, object2, deadlyItems,
        friendlyItem);

// Method to set the scrolling speed.
scrollScreen();

// Assigns farthest travel to score.
playerone.increaseScore(playerone.getMovement());

playertwo.increaseScore(playertwo.getMovement());

// Redraws sprites on screen.
repaint();
```

Figure 25: Map Scrolling for the Game

Libraries and APIs

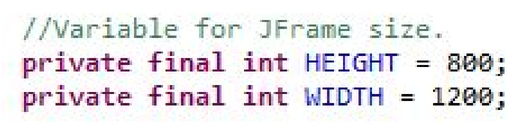
For this project, we incorporate the use of Java Swing, Java Abstract Windowing Toolkit(AWT), Java URL and Java Sound Technology to assist in the development of the game.

A. <u>Java Swing</u>

Swing is a toolkit that provides graphical user interface for Java programs. The classes used from the Swing library are listed below.

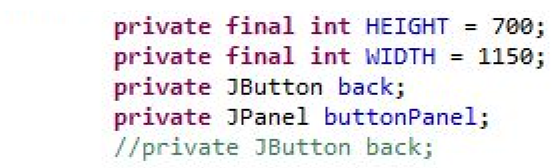
JFrame = A window that contains borders, title and can support button components from Swing.

In the FlashyRoad class, JFrame is used to specify the dimensions of the playing screen which are set with the value of 800 for its height and 1200 for its width, respectively. Meanwhile, the window size for the game instructions and leaderboard screens are slightly smaller in size. The dimensions for the respective screens are set to the value of 700 for its height and width of value 1150.



```java
//Variable for JFrame size.
private final int HEIGHT = 800;
private final int WIDTH = 1200;
```

Figure 26: JFrame for Flashyroad playing screen



```java
private final int HEIGHT = 700;
private final int WIDTH = 1150;
private JButton back;
private JPanel buttonPanel;
//private JButton back;
```

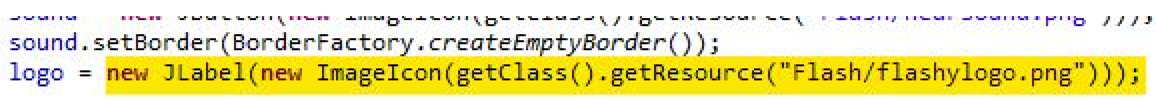
Figure 27 : JFrame for the game instruction screen

JOptionPane = Makes it easy to pop up a standard dialog box.

In the Game class, JOptionPane is used in the switch statement upon the player's death. When the player dies, the box will pop up and display the dialog as shown in Figure 7.

JLabel = A display area for a short text string or an image, or both.

JLabel is used in the Game class to create the logo of the game.

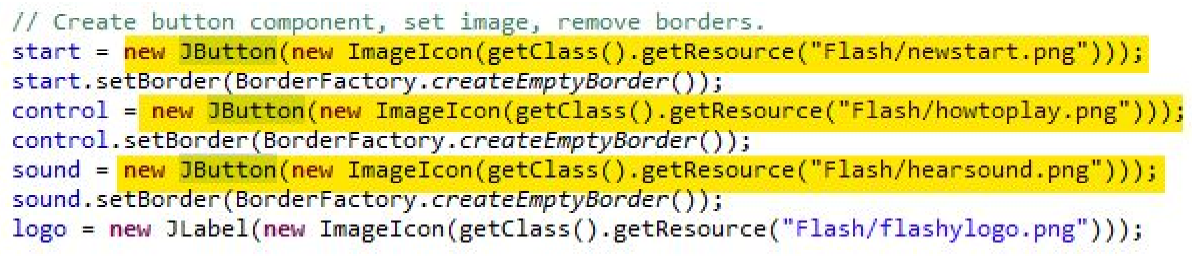


```java
sound.setBorder(BorderFactory.createEmptyBorder());
logo = new JLabel(new ImageIcon(getClass().getResource("Flash/flashylogo.png")));
```

Figure 28: JLabel used to create the logo of the game.

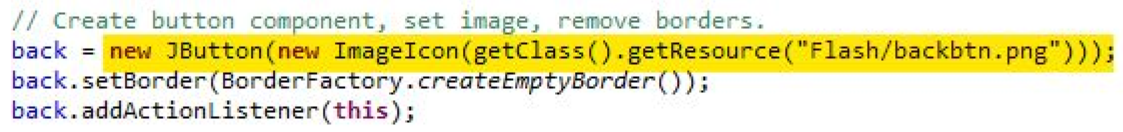
JButton = An implementation of a "push" button.

JButton is used in the Game class to create the "START" button, the "HOW TO PLAY" button which will start the game and show the game instruction when pressed. It is also used to display the icon to mute the game's audio. In the InstructionWindow class, the JButton is used to exit the game instruction screen and return back to the home screen.



```
// Create button component, set image, remove borders.
start = new JButton(new ImageIcon(getClass().getResource("Flash/newstart.png")));
start.setBorder(BorderFactory.createEmptyBorder());
control = new JButton(new ImageIcon(getClass().getResource("Flash/howtoplay.png")));
control.setBorder(BorderFactory.createEmptyBorder());
sound = new JButton(new ImageIcon(getClass().getResource("Flash/hearsound.png")));
sound.setBorder(BorderFactory.createEmptyBorder());
logo = new JLabel(new ImageIcon(getClass().getResource("Flash/flashylogo.png")));
```

Figure 29: JButton used in Game class

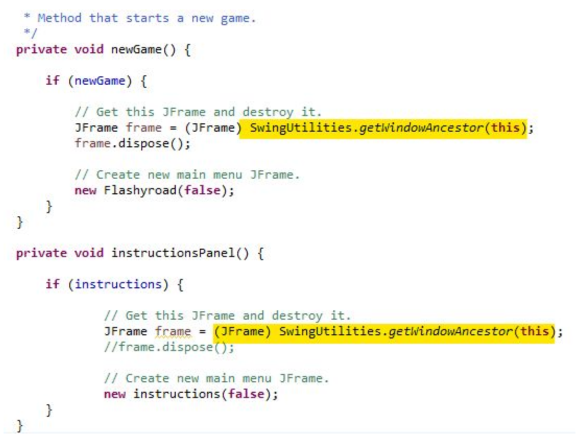


```
// Create button component, set image, remove borders.
back = new JButton(new ImageIcon(getClass().getResource("Flash/backbtn.png")));
back.setBorder(BorderFactory.createEmptyBorder());
back.addActionListener(this);
```

Figure 30: JButton used in InstructionWindow class.

SwingUtilities = A collection of utility methods for Swing

The utility methods used in the SwingUtilities class is the getWindowAncestor in the Game class. This will remove and dispose of the current screen if not needed, conserving resources.



```
 * Method that starts a new game.
 */
private void newGame() {

    if (newGame) {

        // Get this JFrame and destroy it.
        JFrame frame = (JFrame) SwingUtilities.getWindowAncestor(this);
        frame.dispose();

        // Create new main menu JFrame.
        new Flashyroad(false);
    }
}

private void instructionsPanel() {

    if (instructions) {

            // Get this JFrame and destroy it.
            JFrame frame = (JFrame) SwingUtilities.getWindowAncestor(this);
            //frame.dispose();

            // Create new main menu JFrame.
            new instructions(false);
    }
}
```

Figure 31: SwingUtilities class used in Game class

ImageIcon = An implementation of the Icon interface that paints Icons from Images

In figure 32, this class is used together with the JButton class. This allows the buttons to have a custom image so the buttons can be differentiated from one another. In our project, we used four different images to label the START button, the HOW TO PLAY button and LEADERBOARD button in the main screen and BACK button in the game instruction screen.

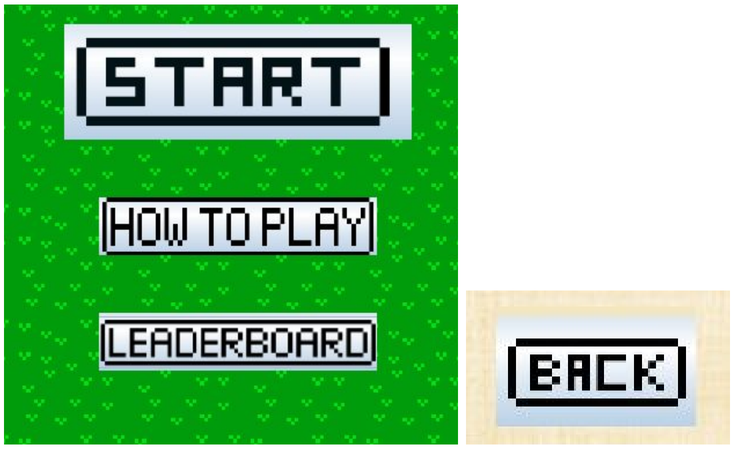


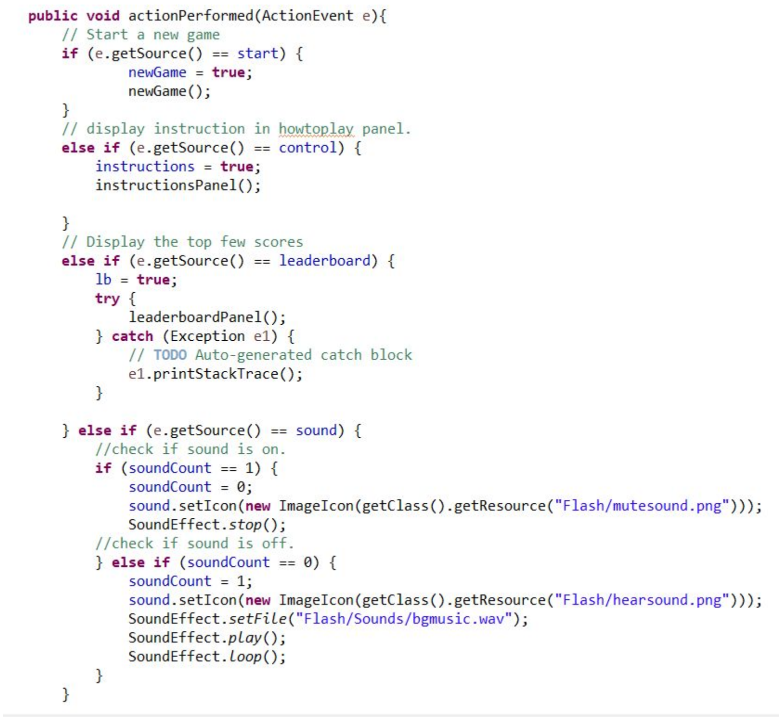
Figure 32: Image icon of the buttons.

BorderFactory = Factory class for vending standard Border objects.

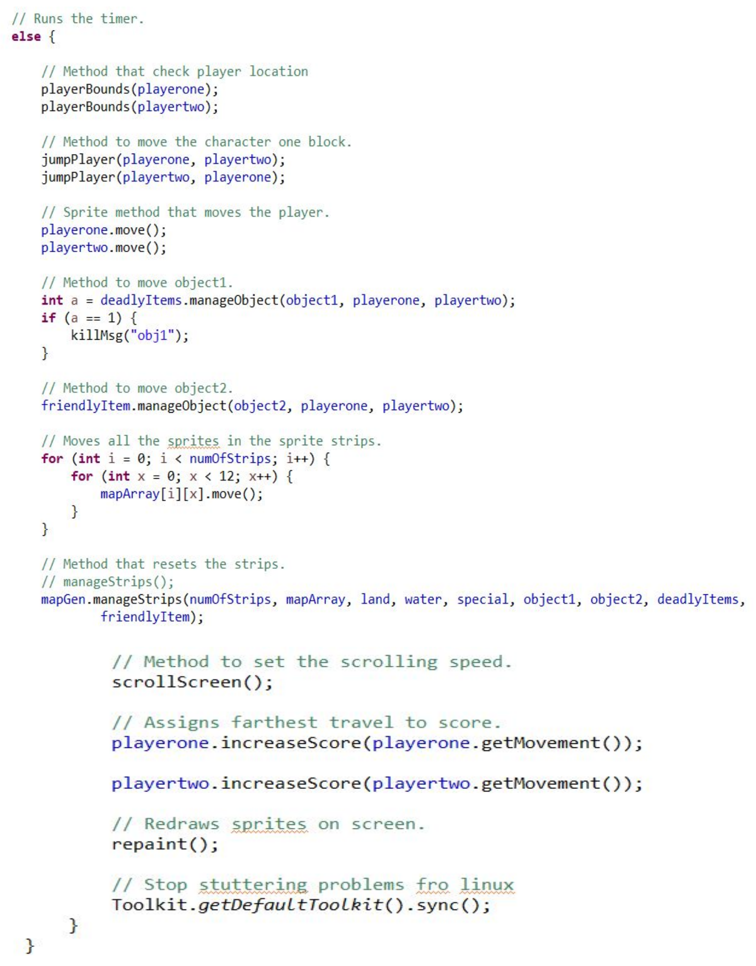
Using the createEmptyBorder method in the BorderFactory class to create an empty border that takes up no space for the buttons in this game.

B. Java AWT

ActionEvent = An ActionEvent is an event that occurs.In our case, pressing the button will perform its intended task such as starting the game or muting the music.



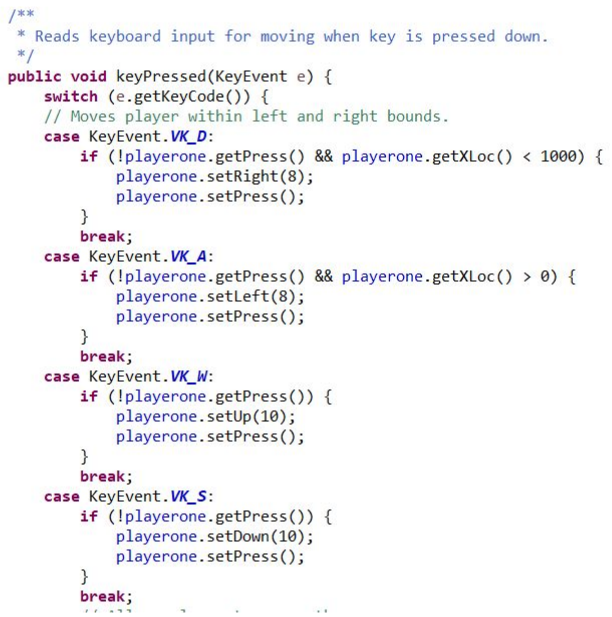
```
public void actionPerformed(ActionEvent e){
    // Start a new game
    if (e.getSource() == start) {
            newGame = true;
            newGame();
    }
    // display instruction in howtoplay panel.
    else if (e.getSource() == control) {
        instructions = true;
        instructionsPanel();

    }
    // Display the top few scores
    else if (e.getSource() == leaderboard) {
        lb = true;
        try {
            leaderboardPanel();
        } catch (Exception e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }

    } else if (e.getSource() == sound) {
        //check if sound is on.
        if (soundCount == 1) {
            soundCount = 0;
            sound.setIcon(new ImageIcon(getClass().getResource("Flash/mutesound.png")));
            SoundEffect.stop();
        //check if sound is off.
        } else if (soundCount == 0) {
            soundCount = 1;
            sound.setIcon(new ImageIcon(getClass().getResource("Flash/hearsound.png")));
            SoundEffect.setFile("Flash/Sounds/bgmusic.wav");
            SoundEffect.play();
            SoundEffect.loop();
        }
    }
```



```java
// Runs the timer.
else {

    // Method that check player location
    playerBounds(playerone);
    playerBounds(playertwo);

    // Method to move the character one block.
    jumpPlayer(playerone, playertwo);
    jumpPlayer(playertwo, playerone);

    // Sprite method that moves the player.
    playerone.move();
    playertwo.move();

    // Method to move object1.
    int a = deadlyItems.manageObject(object1, playerone, playertwo);
    if (a == 1) {
        killMsg("obj1");
    }

    // Method to move object2.
    friendlyItem.manageObject(object2, playerone, playertwo);

    // Moves all the sprites in the sprite strips.
    for (int i = 0; i < numOfStrips; i++) {
        for (int x = 0; x < 12; x++) {
            mapArray[i][x].move();
        }
    }

    // Method that resets the strips.
    // manageStrips();
    mapGen.manageStrips(numOfStrips, mapArray, land, water, special, object1, object2, deadlyItems,
            friendlyItem);

    // Method to set the scrolling speed.
    scrollScreen();

    // Assigns farthest travel to score.
    playerone.increaseScore(playerone.getMovement());

    playertwo.increaseScore(playertwo.getMovement());

    // Redraws sprites on screen.
    repaint();

    // Stop stuttering problems fro linux
    Toolkit.getDefaultToolkit().sync();
}
}
```

Figure 33: Code Snippet of ActionEvent

KeyAdapter = A class to receive keyboard events.
KeyEvent = An event which indicates that a keystroke occurred in a component.
KeyListener = The listener interface for receiving keystrokes.
In our project, when the players press the W,A,S,D or up, down, left, right arrow keys will move the individual character. Pressing the shift key will pause the game.



```java
/**
 * Reads keyboard input for moving when key is pressed down.
 */
public void keyPressed(KeyEvent e) {
    switch (e.getKeyCode()) {
    // Moves player within left and right bounds.
    case KeyEvent.VK_D:
        if (!playerone.getPress() && playerone.getXLoc() < 1000) {
            playerone.setRight(8);
            playerone.setPress();
        }
        break;
    case KeyEvent.VK_A:
        if (!playerone.getPress() && playerone.getXLoc() > 0) {
            playerone.setLeft(8);
            playerone.setPress();
        }
        break;
    case KeyEvent.VK_W:
        if (!playerone.getPress()) {
            playerone.setUp(10);
            playerone.setPress();
        }
        break;
    case KeyEvent.VK_S:
        if (!playerone.getPress()) {
            playerone.setDown(10);
            playerone.setPress();
        }
        break;
```

Figure 34: How keystrokes are registered and triggered player one's movement

C. Java Sound API

The javax.sound.sampled package allows recording and playback of audio files. In our project, we used this package to produce music when the game is played.

AudioInputStream = An audio input stream with specific audio format and length

AudioSystem = This class acts as the entry point to the sampled-audio system resources. Using the getAudioInputStream method from this class to obtain an audio input stream.

Clip = This interface allows audio data to be loaded before playback instead of being streamed in real time.

Strengths and Limitations

A. Strengths

The project focused on the functions and gameplay mechanics allowing different objects to interact differently with each other. These methods are tried and tested through trials and errors and developed based on the programming knowledge we have of Java. Debugging and testing is constantly done throughout the development of the game to find any faults or errors which were not caught by the error handling.

With a strong game logic and the implementation of objects and classes the project has a robust framework which reduces the amount of redundancies and increases the ease of maintenance. An example is that obstacles and characters inherit sprite classes which allows methods for storing the image and position of the objects, making it easier to read and maintain the codes.

B. Limitations

The future improvement of the game is to increase the difficulty as the game progresses using the game score. This feature is not easy to implement as the timer class is an immutable object — when the game is running and other game logic has to be implemented or more time will be used to implement different difficulty levels.

Another future improvement is for the game to be executed as an executable file, making it easier to run the java file without having to use an integrated development environment (IDE). This would allow better concealing of the codes as the executable file will run immediately.

The limitation of FlashyRoad is the success path generation, we could not identify a middle ground where a success path is always generated to give the player a perfect route. However we did manage to achieve a dynamic map generation of the obstacles and items to let users have a different experience / difficulty every time they play FlashyRoad.

Team Contributions

Name	Contribution
ASIF	● Collision prevention between 2 players. ● Abstract for Obstacle/ Item classes. ● Obstacle / Item classes and subclasses logic. ● Rendering of game map. ● Debugging and testing. ● Report
HOU LIANG	● Movement of obstacles. ● Logic for scoring system. ● Collision detection for objects and obstacles. ● 2 Interfaces hit sound and object size checker. ● Sound effects and mute option. ● Report
NAZRI	● Start and Pause Screen. ● Life system. ● Debugging & Testing. ● Report
TERENCE	● Implemented multiplayer feature, key listener and action listener. ● Implemented map generator. ● Implemented read/write to files for scores and scoreboard. ● Debugging & Testing. ● Report
AIN	● Rendering of main menu GUI ● Game Instruction panel ● Instruction panel. ● Debugging and testing ● Report ● Video Editing
ZI YING	● Rendering of sprites, obstacles and characters ● Background music ● Movement of sprite. ● Debugging and testing ● Report ● Video Editing

Reflections

Name	Reflections
ASIF	As the project comes to an end, I strongly feel that my understanding of OOP concepts has been solidified. I have learnt that object-oriented programming allows codes to be modular, flexible and allows reusability. An example will be the use of Inheritance. When learning just the concepts, I honestly felt that it is nothing much. But when it came to coding a big project like the game, I realised how efficient the codes are with Inheritance. Not only Inheritance, but with Abstract, Polymorphism, and Interfaces. With Abstract and Polymorphism. I learnt that I could group subclasses together to be sibling classes. With that, I can implement overriding or overloading methods on each class. Which keeps the code structure organised and understandable.

HOU LIANG	After finishing this project, it helped me to understand Object Oriented Programming concepts (OOP) better. Polymorphism provides the useful ability to combine methods of different classes. This is useful not only as the amount of code is lesser but the overriding and overwriting allows us to modify the child class to suit the types of scenarios. We can also call separate object properties using the same classes to access different objects. Inheritance allows us to create child classes that inherit the methods of another parent class. Inheriting methods and functions from parents objects are useful and conserves assets as there is no duplication of the same methods. Abstract classes allow us to hide high level functions to provide better readability of codes. Interfaces allow two or more different classes to share a similar method but they are able to interpret the method differently. Encapsulation and Error handling are features that provide better security of the code and structure, making sure that our code does not run into errors that are not predetermined. Furthermore, encapsulation determines the boundaries to ensure that users do not need to understand the game to play it. Classes are really useful overall as they are the templates to create different types of objects. Compared to procedural, OOP is easier to incorporate new data and functions, making the codes aesthetically pleasing.
NAZRI	Developing a game using OOP uses less development time since codes can be reused and create a model based on objects. Using OOP, the game can be broken down into smaller objects for easier troubleshooting.
TERENCE	Translating the object oriented programming concepts we learned in the lab to our own project was tedious at first. However, as we delved into starting to create the classes for our game, the concepts became perceivable and became simpler to implement and understand. However on the portion of the interface, it still seems a little bit dubious to some of us as it took us quite a while to implement the interfaces without trying to force the concepts. Overall, this java project on creating a game was enriching as we learn to work with the object oriented design pattern and the java libraries such as swing.

AIN	At first, before attempting the project, developing a game seems like a very far-fetched thing for me. But after learning the OOP concepts and applying it to our project, I realised how these concepts are able to help us to develop our game efficiently. Without the OOP concepts, our game codes will not be as organised and all will be messy in a file. I am also thankful for my teammates that help me out with the codes whenever I face some problems during the development.
ZI YING	Before doing the project,I did not know how to apply the concepts of the OOP into our project. However, with the guidance of my teammates, I slowly understood why we needed to use OOP in our codes as it made our codes look neater for not only programmers but also for others who are looking at the codes. Even though I am not strong in coding, my teammates would help me when I was stuck with certain codes and they would also patiently teach me some parts of the codes.