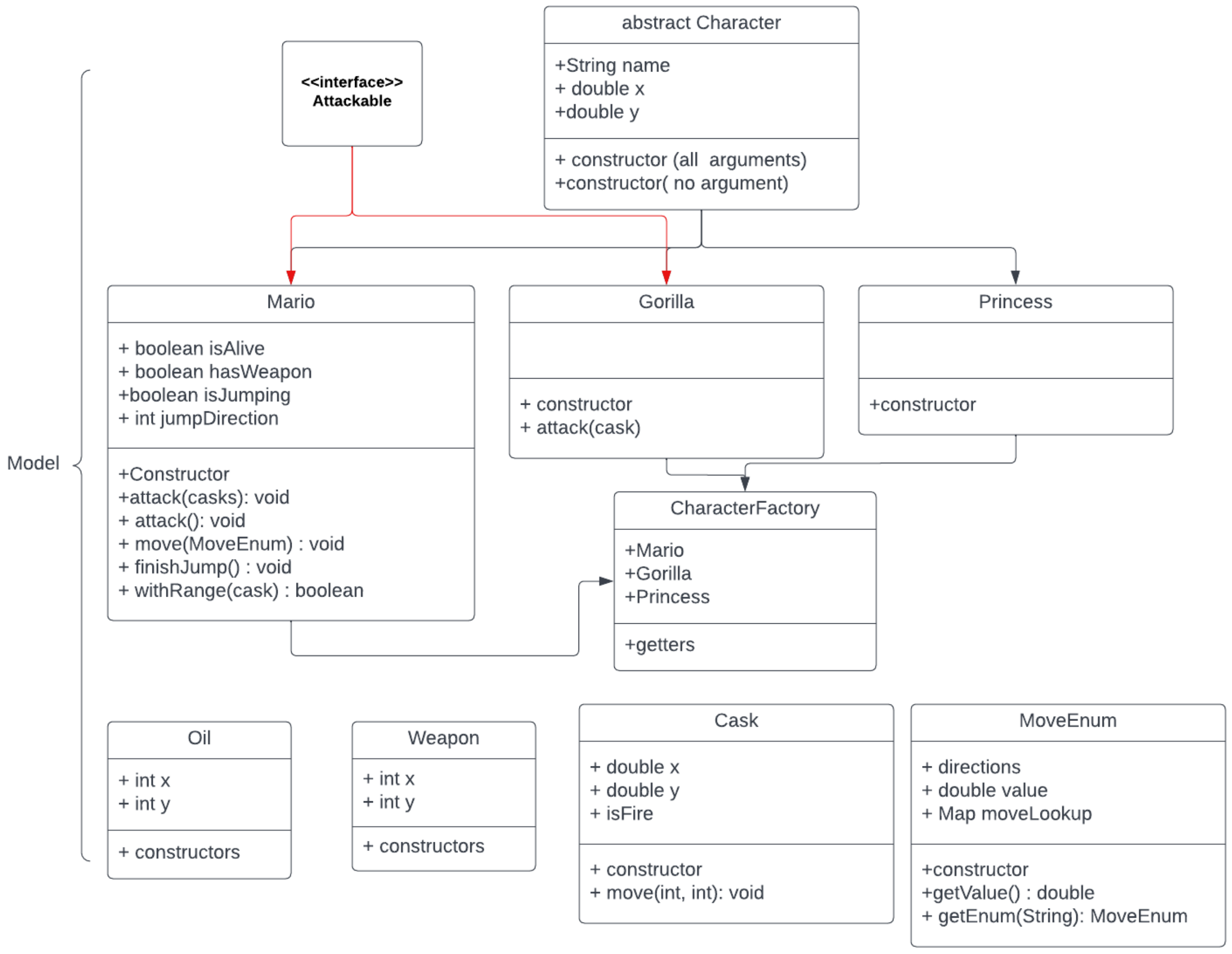
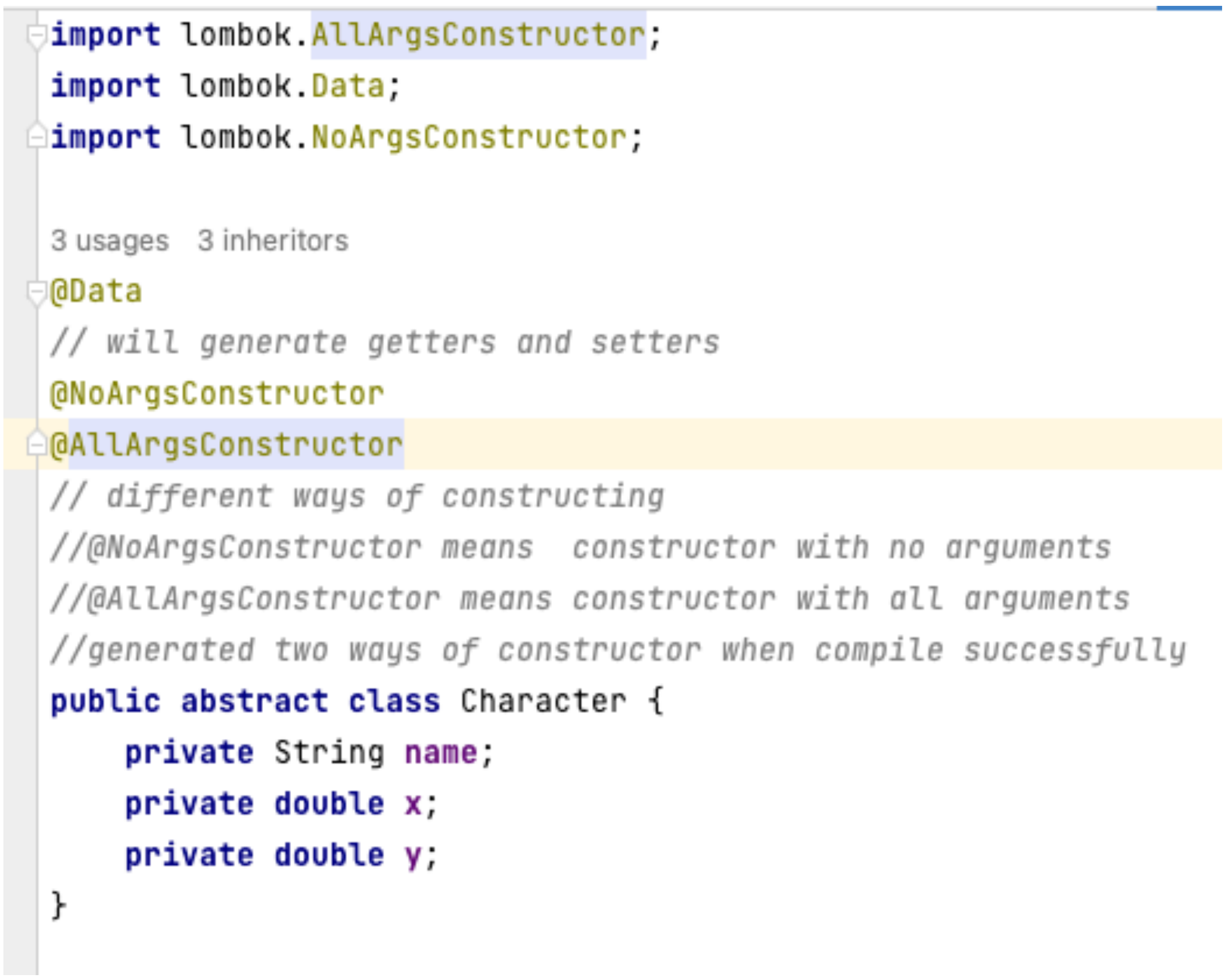
Donkey Kong design

Author : Xiaonan Liu

I will explain my code with class diagram



In the model
Base:
I define an interface named Attackable, it only has a function named attack, and the parameter – List of casks with void return value
Also, a parent class named Character, the field is showing above. (Constructors are auto generate from Lombok, also all class have getters and setters ,also will auto generate by Lombok)

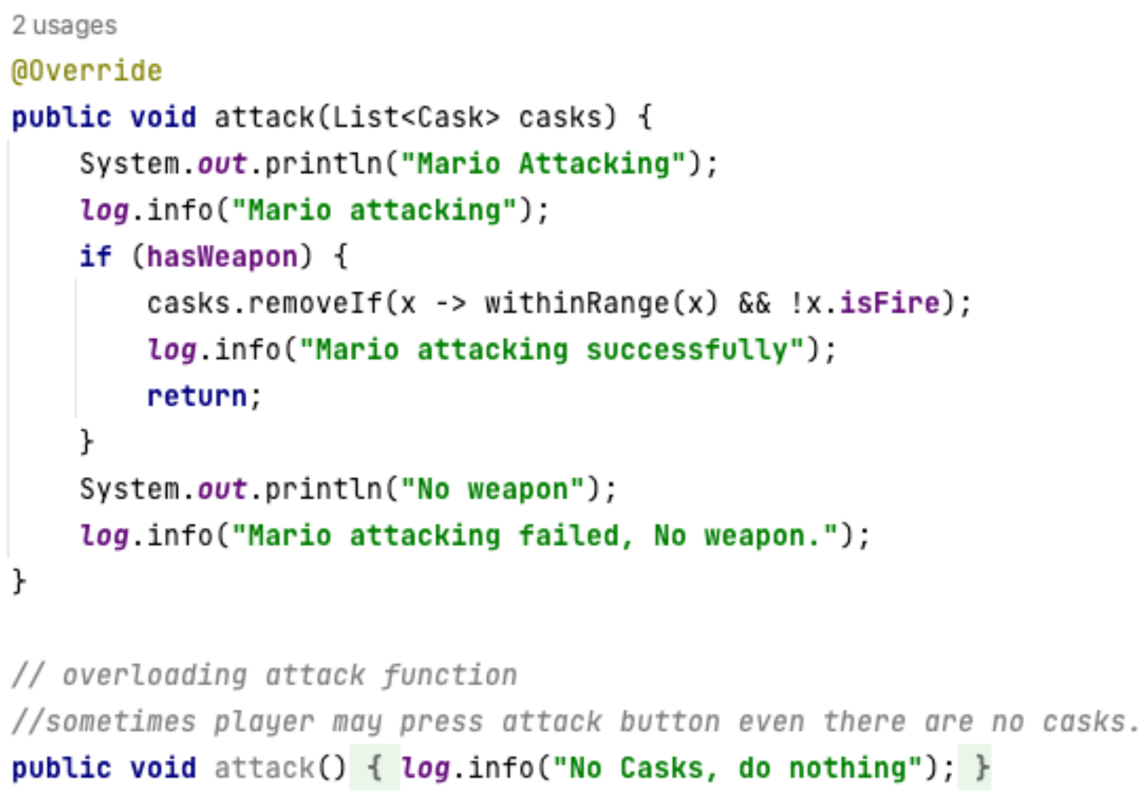


```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

3 usages   3 inheritors
@Data
// will generate getters and setters
@NoArgsConstructor
@AllArgsConstructor
// different ways of constructing
//@NoArgsConstructor means  constructor with no arguments
//@AllArgsConstructor means constructor with all arguments
//generated two ways of constructor when compile successfully
public abstract class Character {
    private String name;
    private double x;
    private double y;
}
```

Mario class, Gorilla class, and Princess class are all inheritance from Character class, and Mario class and Gorilla class will also inheritance Attackable.

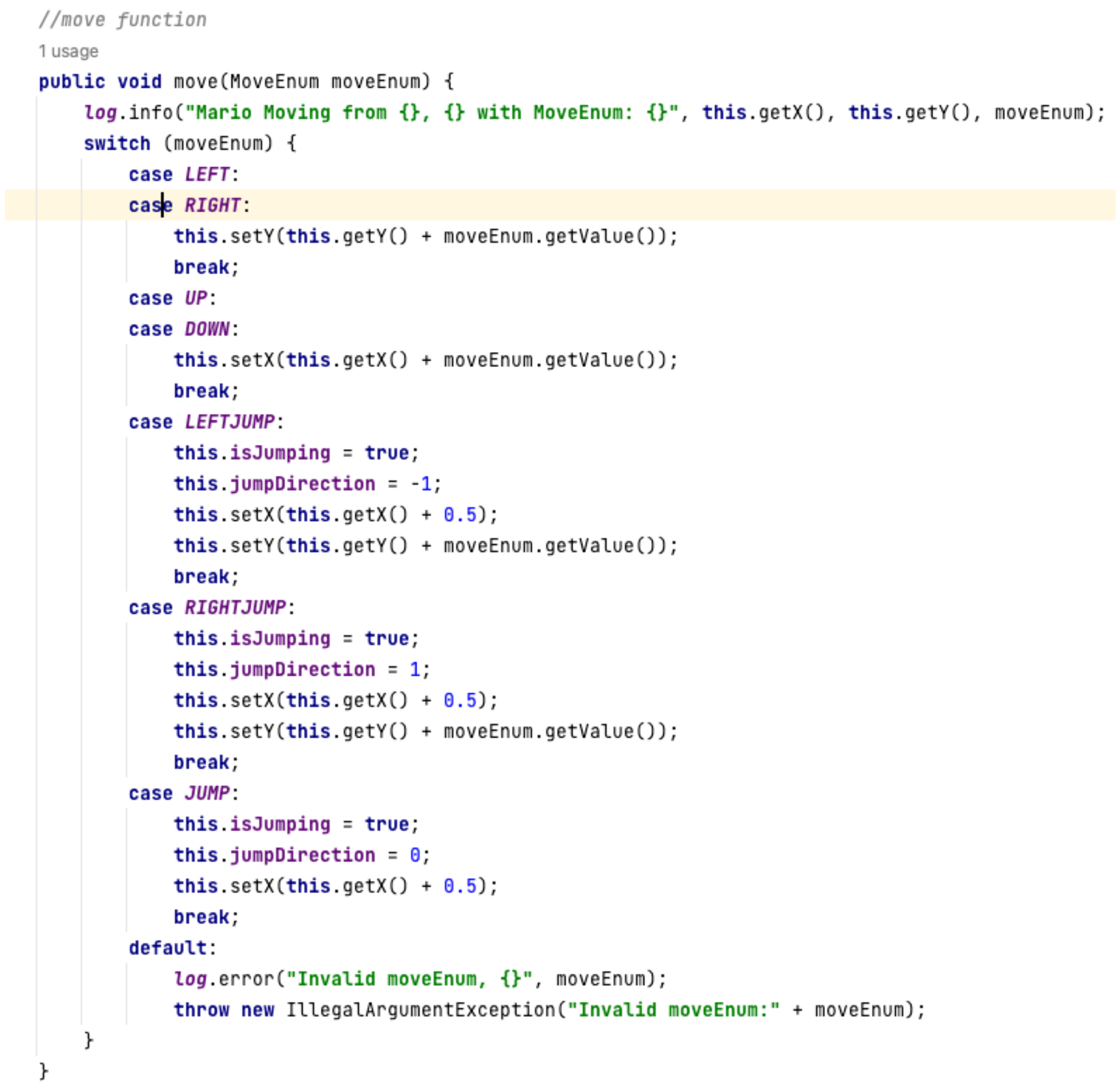
The child class will inheritance all the fields and the functions from the parent class, so I will only analysis their unique functions, or the functions I overrided or overloaded.

In Mario class, I will override attack function, if Mario has weapon and cask is in Mario’s attack range, Mario will attack successfully, and this cask will remove from our list, also I overload the attack function will no parameters, because sometimes plays pressed attack but there are no casks right now.



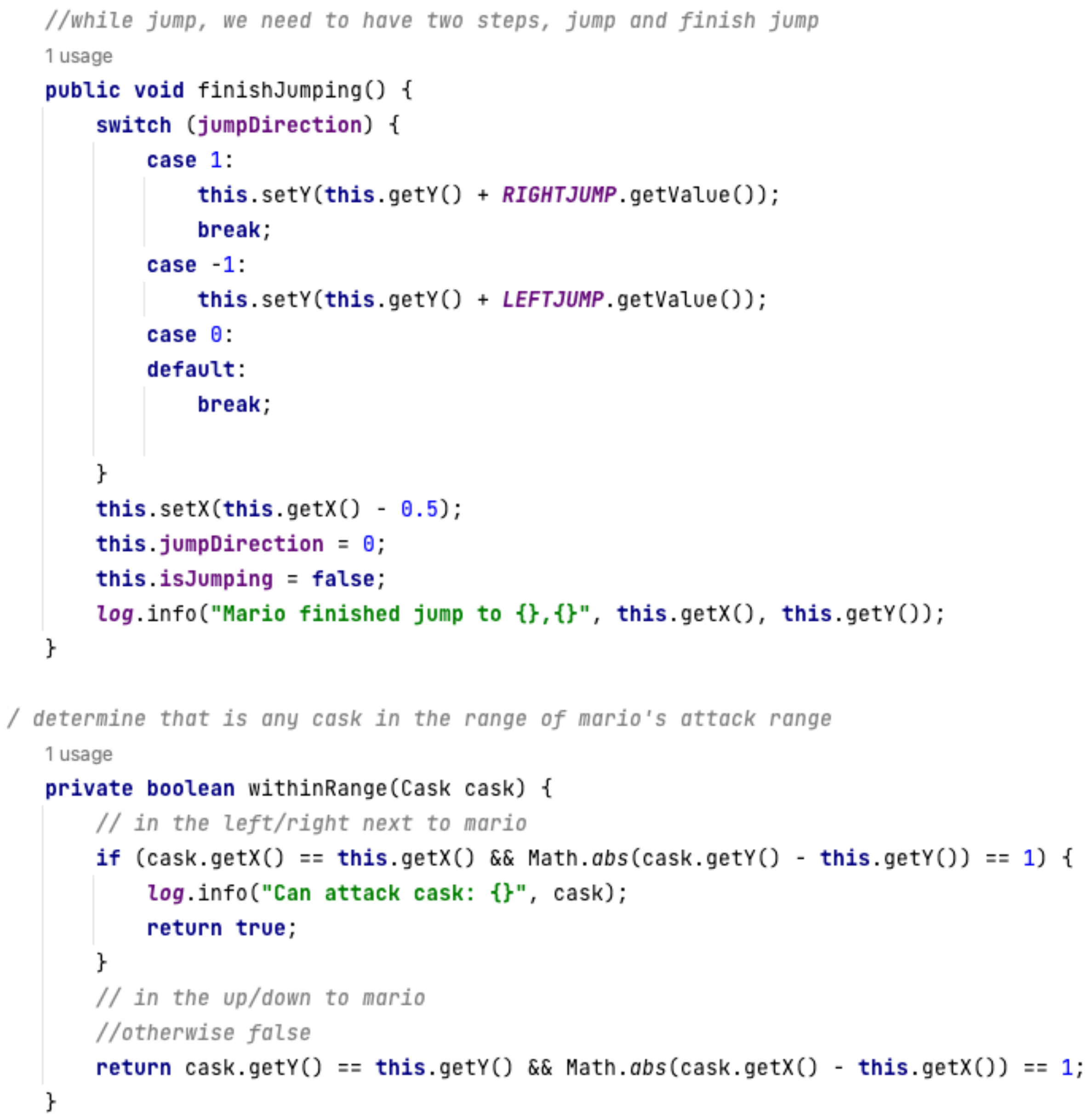
```
2 usages
@Override
public void attack(List<Cask> casks) {
    System.out.println("Mario Attacking");
    log.info("Mario attacking");
    if (hasWeapon) {
        casks.removeIf(x -> withinRange(x) && !x.isFire);
        log.info("Mario attacking successfully");
        return;
    }
    System.out.println("No weapon");
    log.info("Mario attacking failed, No weapon.");
}

// overloading attack function
//sometimes player may press attack button even there are no casks.
public void attack() { log.info("No Casks, do nothing"); }
```

I also define a move function for Mario, it will update Mario`s position according to the player`s choices.



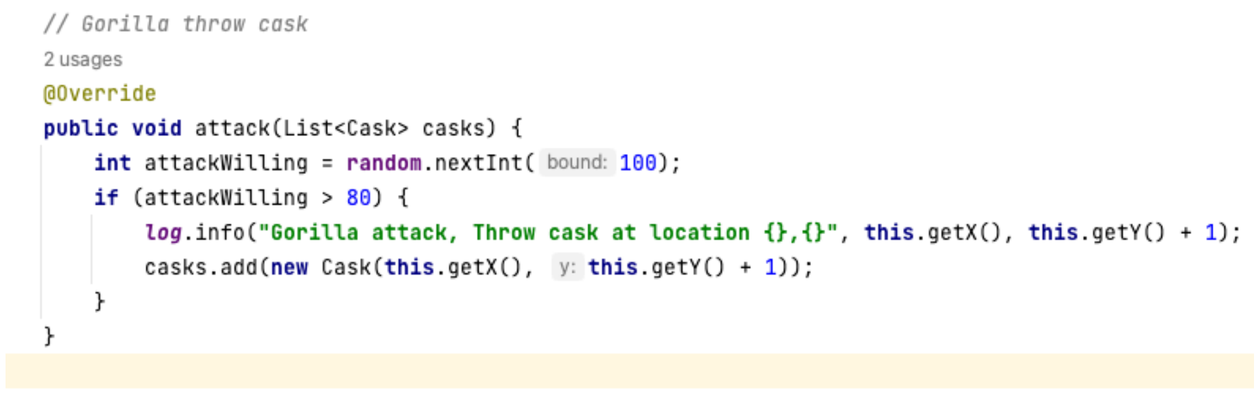
```
//move function
1 usage
public void move(MoveEnum moveEnum) {
    log.info("Mario Moving from {}, {} with MoveEnum: {}", this.getX(), this.getY(), moveEnum);
    switch (moveEnum) {
        case LEFT:
        case RIGHT:
            this.setY(this.getY() + moveEnum.getValue());
            break;
        case UP:
        case DOWN:
            this.setX(this.getX() + moveEnum.getValue());
            break;
        case LEFTJUMP:
            this.isJumping = true;
            this.jumpDirection = -1;
            this.setX(this.getX() + 0.5);
            this.setY(this.getY() + moveEnum.getValue());
            break;
        case RIGHTJUMP:
            this.isJumping = true;
            this.jumpDirection = 1;
            this.setX(this.getX() + 0.5);
            this.setY(this.getY() + moveEnum.getValue());
            break;
        case JUMP:
            this.isJumping = true;
            this.jumpDirection = 0;
            this.setX(this.getX() + 0.5);
            break;
        default:
            log.error("Invalid moveEnum, {}", moveEnum);
            throw new IllegalArgumentException("Invalid moveEnum:" + moveEnum);
    }
}
```

Besides these, I also define two helper functions, one is for jump, first we update the position when Mario is jump, and then we will set back the coordinate to finish jumping. Another help function is to check is the cask in the Mario’s attack range, return true if this is in range, otherwise return false.



```
    //while jump, we need to have two steps, jump and finish jump
    1 usage
    public void finishJumping() {
        switch (jumpDirection) {
            case 1:
                this.setY(this.getY() + RIGHTJUMP.getValue());
                break;
            case -1:
                this.setY(this.getY() + LEFTJUMP.getValue());
            case 0:
            default:
                break;

        }
        this.setX(this.getX() - 0.5);
        this.jumpDirection = 0;
        this.isJumping = false;
        log.info("Mario finished jump to {},{}", this.getX(), this.getY());
    }

/ determine that is any cask in the range of mario's attack range
    1 usage
    private boolean withinRange(Cask cask) {
        // in the left/right next to mario
        if (cask.getX() == this.getX() && Math.abs(cask.getY() - this.getY()) == 1) {
            log.info("Can attack cask: {}", cask);
            return true;
        }
        // in the up/down to mario
        //otherwise false
        return cask.getY() == this.getY() && Math.abs(cask.getX() - this.getX()) == 1;
    }
```

In Gorilla class
 We also override the attack function, we will generate a random number, if it is over 80, gorilla will throw a cask, and the cask's position will be add in the list of casks.

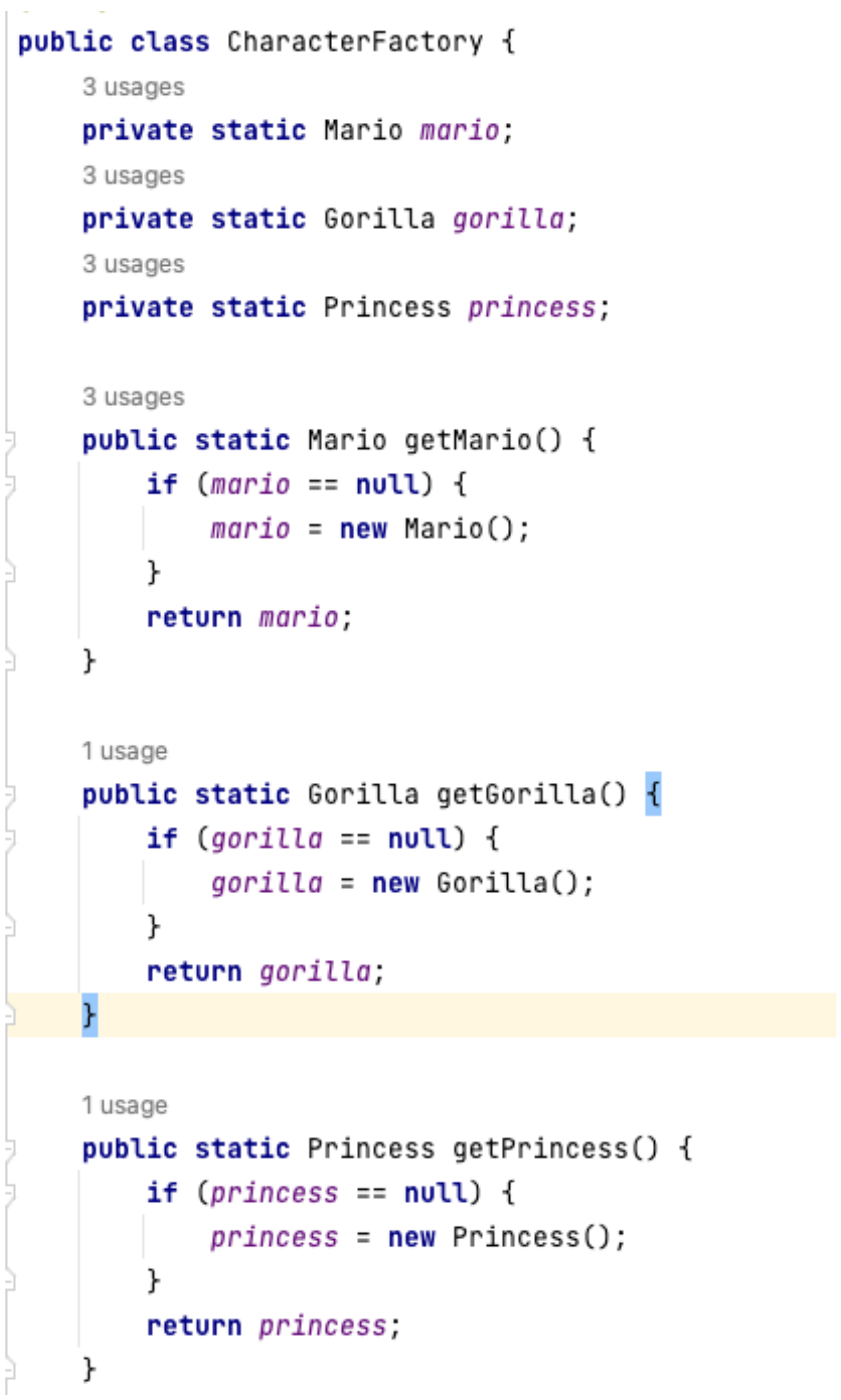


```java
// Gorilla throw cask
2 usages
@Override
public void attack(List<Cask> casks) {
    int attackWilling = random.nextInt( bound: 100);
    if (attackWilling > 80) {
        log.info("Gorilla attack, Throw cask at location {},{}", this.getX(), this.getY() + 1);
        casks.add(new Cask(this.getX(),  y: this.getY() + 1));
    }
}
```

Princess class only has constructors and all field, functions inheritance from

Character class

CharacterFactory class is defined for use singleton model, we will generate our Mario, Gorilla, and Princess when we have an instance of this class, and it will make sure we will only have only one of them during the game. So, for this class, we have 3 variables, Mario, Gorilla and Princess, and getters for them.

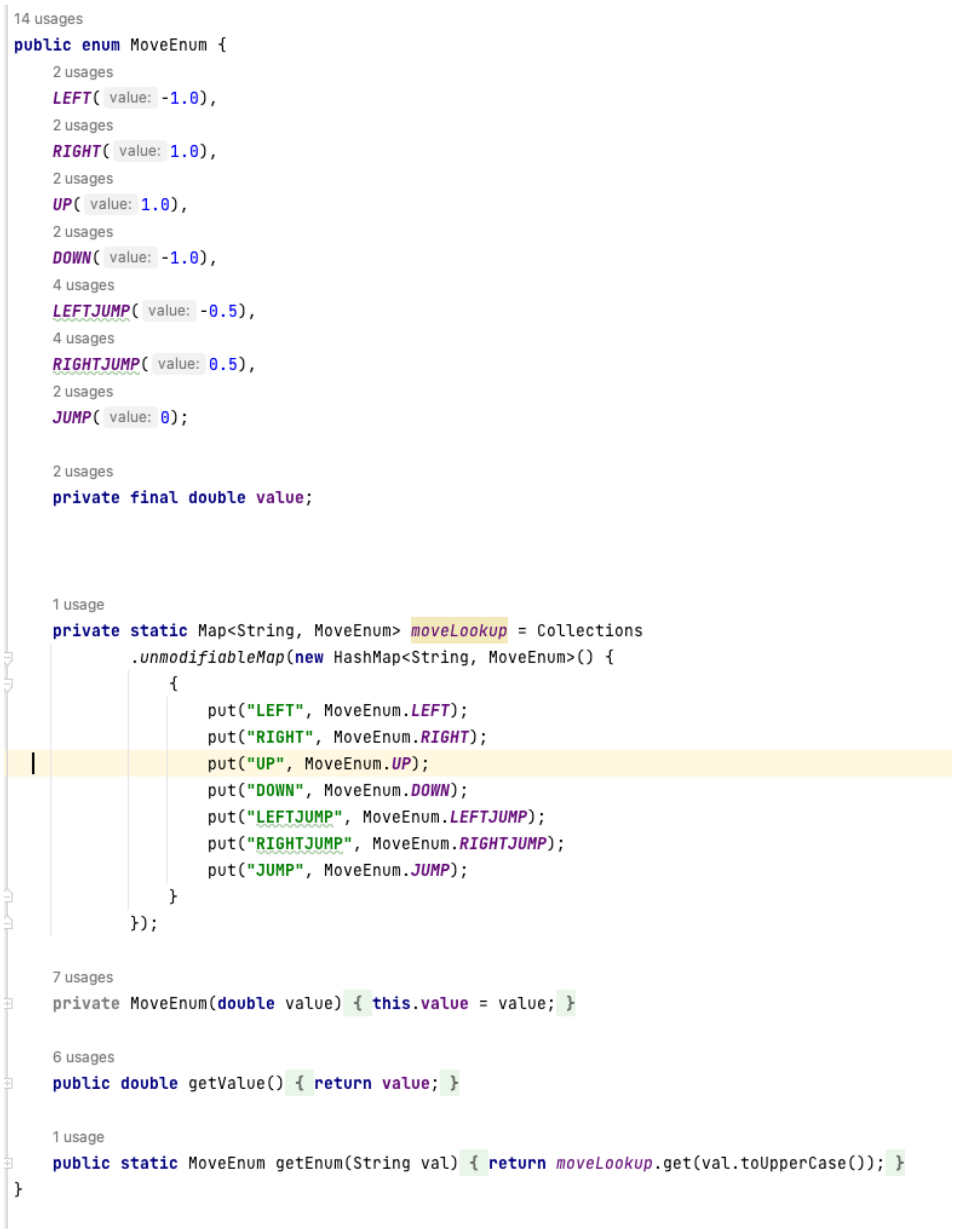


```
public class CharacterFactory {
    3 usages
    private static Mario mario;
    3 usages
    private static Gorilla gorilla;
    3 usages
    private static Princess princess;

    3 usages
    public static Mario getMario() {
        if (mario == null) {
            mario = new Mario();
        }
        return mario;
    }

    1 usage
    public static Gorilla getGorilla() {
        if (gorilla == null) {
            gorilla = new Gorilla();
        }
        return gorilla;
    }

    1 usage
    public static Princess getPrincess() {
        if (princess == null) {
            princess = new Princess();
        }
        return princess;
    }
```

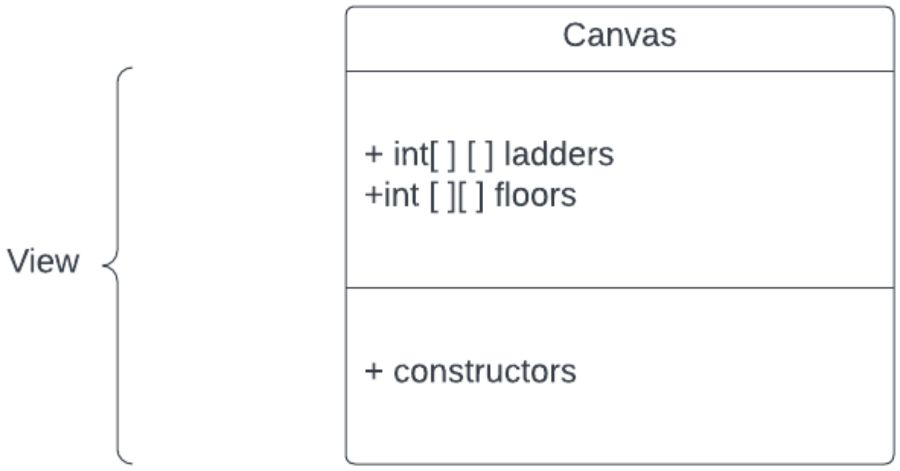
Cask class:
It has 3 variables for its position and a Boolean value to check is it fire, and it will set by false with a new cask instanced was defined, and it also has the move function to update its position, the setters and getters will auto generate by Lombok with the keyword @Data

Same with cask class, the Oil class, weapon class also has field for its position, and constructors, getters, setters, generate by Lombok.

MoveEnum class:
This class predefine the type of move and its value, when character and cask make movement, we could get the value for the specific move type, and get its value for update position



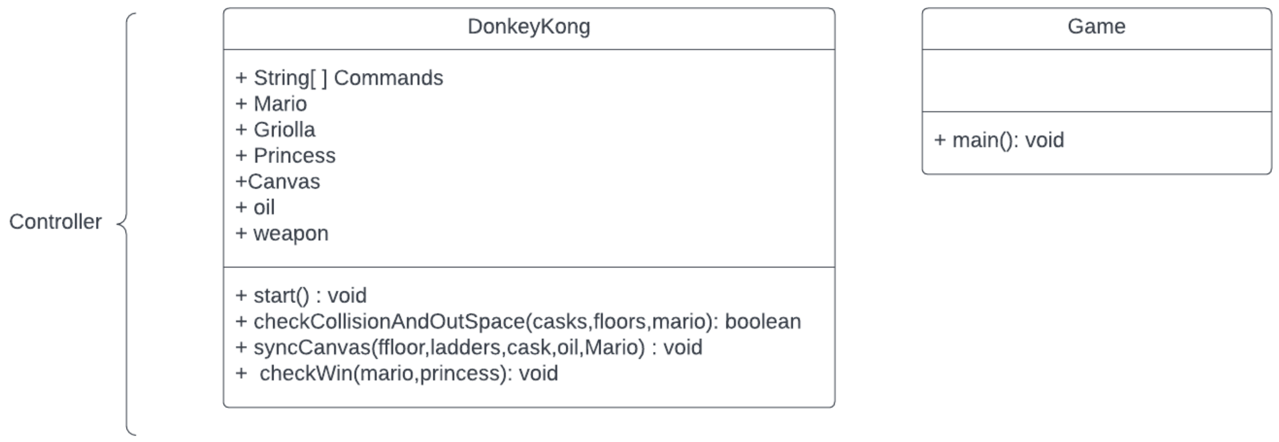
```
14 usages
public enum MoveEnum {
    2 usages
    LEFT( value: -1.0),
    2 usages
    RIGHT( value: 1.0),
    2 usages
    UP( value: 1.0),
    2 usages
    DOWN( value: -1.0),
    4 usages
    LEFTJUMP( value: -0.5),
    4 usages
    RIGHTJUMP( value: 0.5),
    2 usages
    JUMP( value: 0);

    2 usages
    private final double value;



    1 usage
    private static Map<String, MoveEnum> moveLookup = Collections
            .unmodifiableMap(new HashMap<String, MoveEnum>() {
                {
                    put("LEFT", MoveEnum.LEFT);
                    put("RIGHT", MoveEnum.RIGHT);
                    put("UP", MoveEnum.UP);
                    put("DOWN", MoveEnum.DOWN);
                    put("LEFTJUMP", MoveEnum.LEFTJUMP);
                    put("RIGHTJUMP", MoveEnum.RIGHTJUMP);
                    put("JUMP", MoveEnum.JUMP);
                }
            });

    7 usages
    private MoveEnum(double value) { this.value = value; }

    6 usages
    public double getValue() { return value; }

    1 usage
    public static MoveEnum getEnum(String val) { return moveLookup.get(val.toUpperCase()); }
}
```



Canvas class:

This class is the view for this game, it has two maps, one for ladders, one for slopes of the floor, when an instance of Canvas been generated, the screen shows the original view of this game. Also, getters and setters will generate by Lombok.

```java
@Data
public class Canvas {
    // ladder locations
    // 1: ladder's bot
    // 2: ladder's top
    // 3: broken ladder's bot
    // 4: broken ladder's top
    private int[][] ladders;
    // 0: floor with 0 slope
    // 1: floor with 1 slope
    // -1: floor with -1 slope
    // 2: no floor
    private int[][] floors;

    public Canvas(int[][] ladders, int[][] floors, Mario mario, Gorilla
gorilla, Princess princess) {
        this.ladders = ladders;
        this.floors = floors;

        // new Canvas starts
        // init Mario's location
        // [0, 0] is kept for Oil.
        mario.setX(0);
        mario.setY(1);

        gorilla.setX(floors.length - 2);
        gorilla.setY(0);
        princess.setX(floors.length - 1);
        princess.setY(1);

    }
}
```



Controller
DonkeyKong
+ String[ ] Commands
+ Mario
+ Griolla
+ Princess
+Canvas
+ oil
+ weapon
+ start() : void
+ checkCollisionAndOutSpace(casks,floors,mario): boolean
+ syncCanvas(ffloor,ladders,cask,oil,Mario) : void
+ checkWin(mario,princess): void
Game
+ main(): void

DonkeyKong class:

The field includes final static string array for commands, all characters needed for this game, the canvas, a list of casks, the oil and the weapon.

when call the start function, the canvas will be instanced, and the system will scan the input to determined what needs to operation.

While Mario is alive, first we check has we win, if Mario arrive the position of princess , we win, if not we will do other operations.

```
Scanner scanner = new Scanner(System.in);

    //game begins
    while (mario.isAlive()) {
        System.out.println("Mario current location: " + mario.getX() + "," +
mario.getY());
        if (checkWin(mario, princess)) {
            log.info("Mario win");
            System.out.println("Mario win");
            break;
        }
        if (mario.isJumping()) {
            mario.finishJumping();
        } else {
            // check if floor is empty, if so mario will fall
            if (floors[(int) mario.getX()][(int) mario.getY()] == 2) {
                log.info("Empty floor, Mario falling.");
                mario.setX(mario.getX() - 1);
            } else {
                System.out.println("Enter Move Command\n0: Attack: \n1:
Left\n2: Right\n3: Up\n4: Down\n5: Jump\n6: LeftJump\n7: RightJump\n8:
ExitGame");
                int commandIndex = scanner.nextInt();
                //check invalid command
                if (commandIndex < 0 || commandIndex > 8) {
                    System.out.println("Invalid commandIndex[0-8]. Please
enter again");
                    continue;
                }
                //exit command
                if (commandIndex == 8) {
                    log.info("User exit game.");
                    System.out.println("Exit game.");
                    break;
                }
                //broken ladders, mario could not climb
                if ((commandIndex == 3 && ladders[(int) mario.getX()][(int)
mario.getY()] != 1) || (commandIndex == 4 && ladders[(int)
mario.getX()][(int) mario.getY()] != 2)) {
                    System.out.println("No ladder here. Please enter again");
                    continue;
                }
                System.out.println("Executing Command: " +
commands[commandIndex]);
                // attack
                if (commandIndex == 0) {
                    mario.attack(casks);
                } else { // move
                    System.out.println("Mario Moving");
                    mario.move(MoveEnum.getEnum(commands[commandIndex]));
                }
            }
        }
        //check whether Mario could get weapon or not
        if (mario.getX() == weapon.getX() && mario.getY() == weapon.getY()) {
            log.info("Mario get weapon.");
            System.out.println("Mario get weapon.");
            mario.setHasWeapon(true);
        }
        //update canvas
        syncCanvas(floors, ladders, casks, oil, mario);

        gorilla.attack(casks);
        // check whether Mario die or not
        if (checkCollisionAndOutSpace(casks, floors, mario)) {
            log.info("Mario die, game over");
            System.out.println("Mario die, game over");
        }
    }
    scanner.close();
}
```

Helper functions are defined below this function, such as : synchronized canvas function t update base on our operation, and check is any collision for Mario1s position or Mario may falls out of bound, it those case, Mario will die, and the game will over, but if Mario gets to Princess`s position, we will win.

Game class:
 This is only for the Main function.