Java Rogue-Like Game

Contents

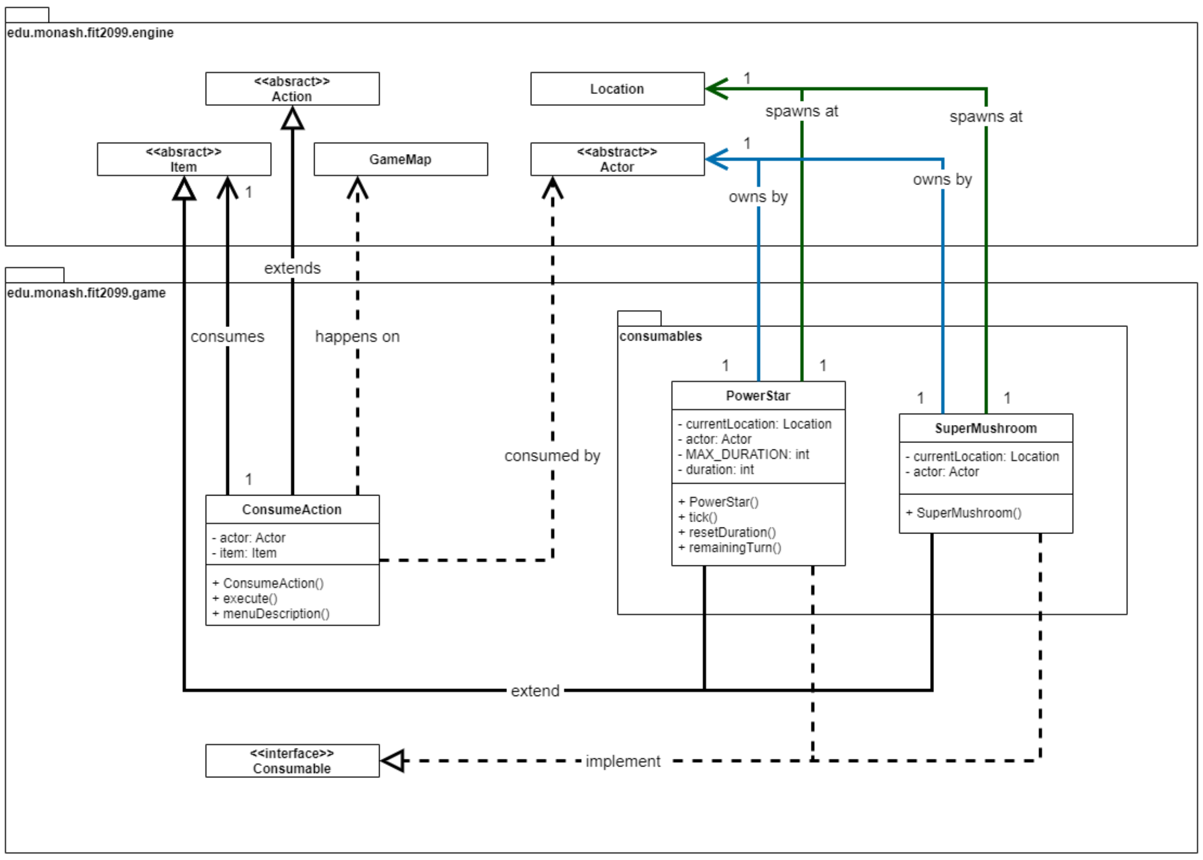
1.0 UML Class Diagrams

This part of the document will include he Unified Modelling Language (UML) class diagrams drawn up for each of the requirements based on their implementation. Any changes made to the previous implementation in Assignment 2 will be highlighted, followed by a brief explanation of what has been changed and the rationale behind the changes.

UML diagrams for all the deliverables, except for REQ1: Lava zone, will be presented. Due to the way this requirement has been implemented, a UML diagram is unnecessary.

1.1 Magical Items

Note: The image may appear small due to the constraints of the document layout. Clarity, and readability of the image may be improved by zooming in.



1.2 More allies and enemies!

Note: The image may appear small due to the constraints of the document layout. Clarity, and readability of the image may be improved by zooming in.

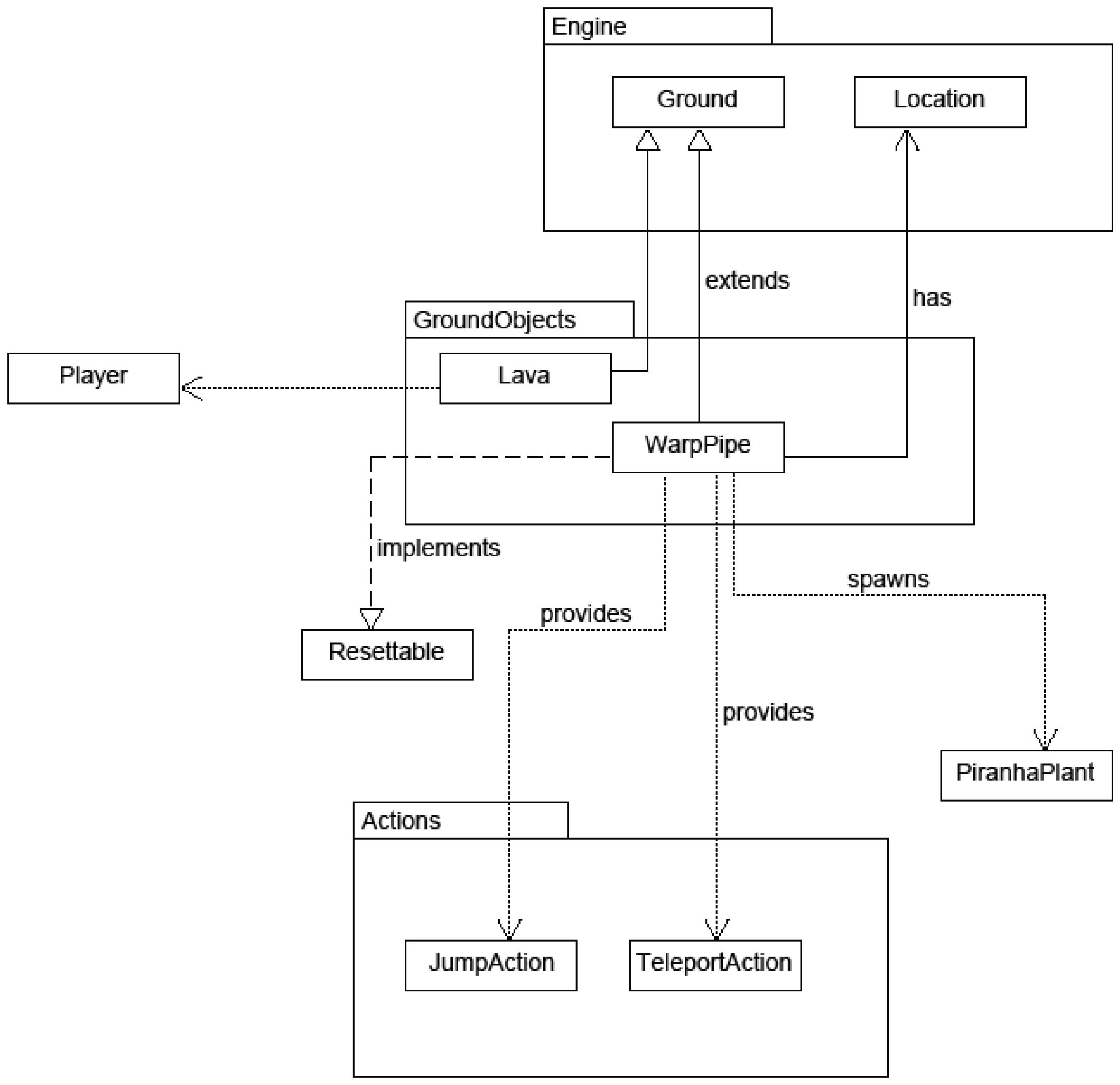


Figure 2 UML diagram for the WarpPipe class.

Note: The image may appear small due to the constraints of the document layout. Clarity, and readability of the image may be improved by zooming in.

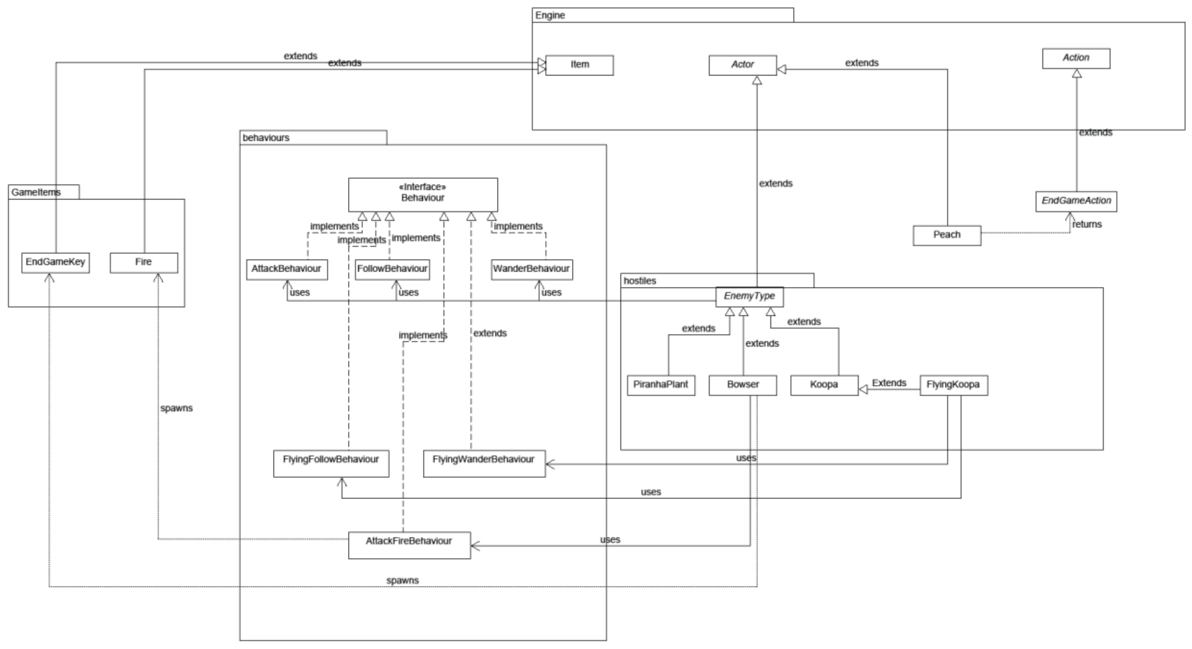


Figure 3 UML diagram for the PiranhaPlant class.

1.3 Magical fountain

Note: The image may appear small due to the constraints of the document layout. Clarity, and readability of the image may be improved by zooming in.

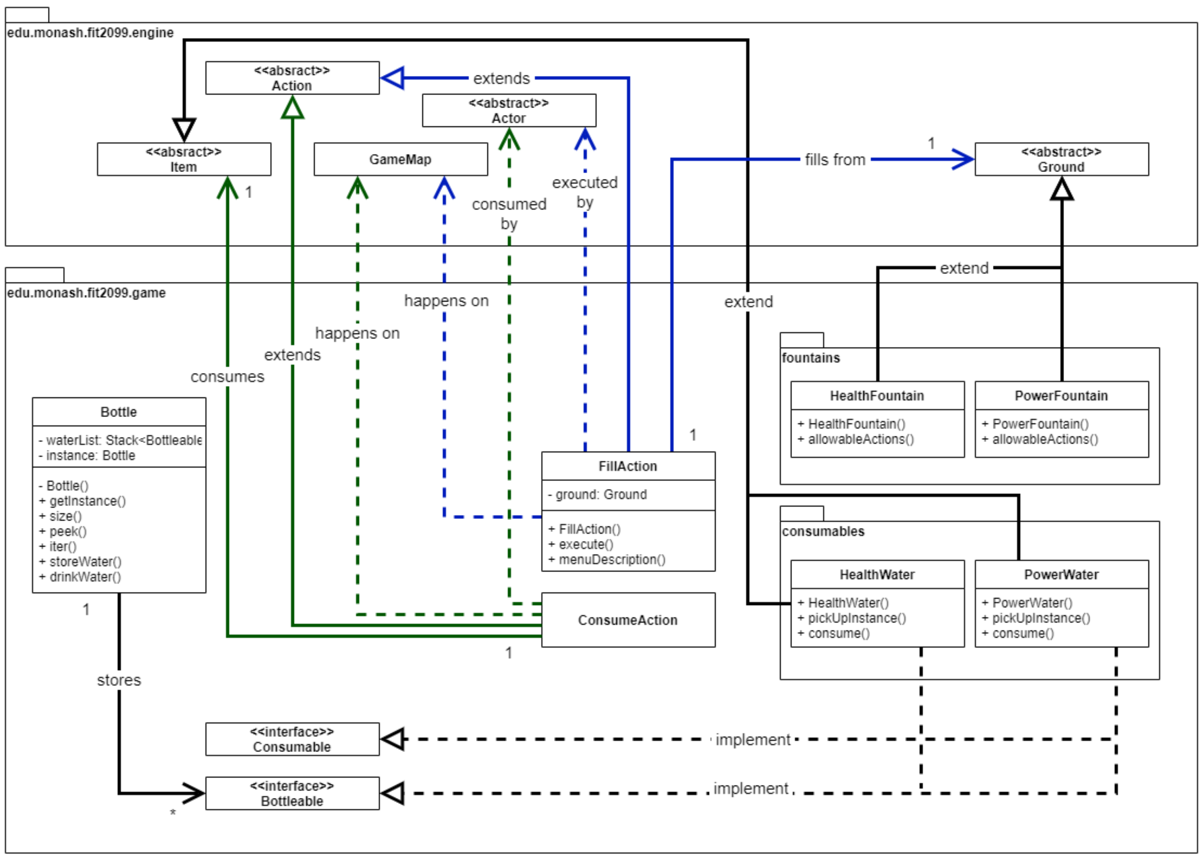


Figure 4 UML diagram for the magical fountains and their respective water, along with the bottle used to store the water.

1.4 Additional debuffs

Note: The image may appear small due to the constraints of the document layout. Clarity, and readability of the image may be improved by zooming in.

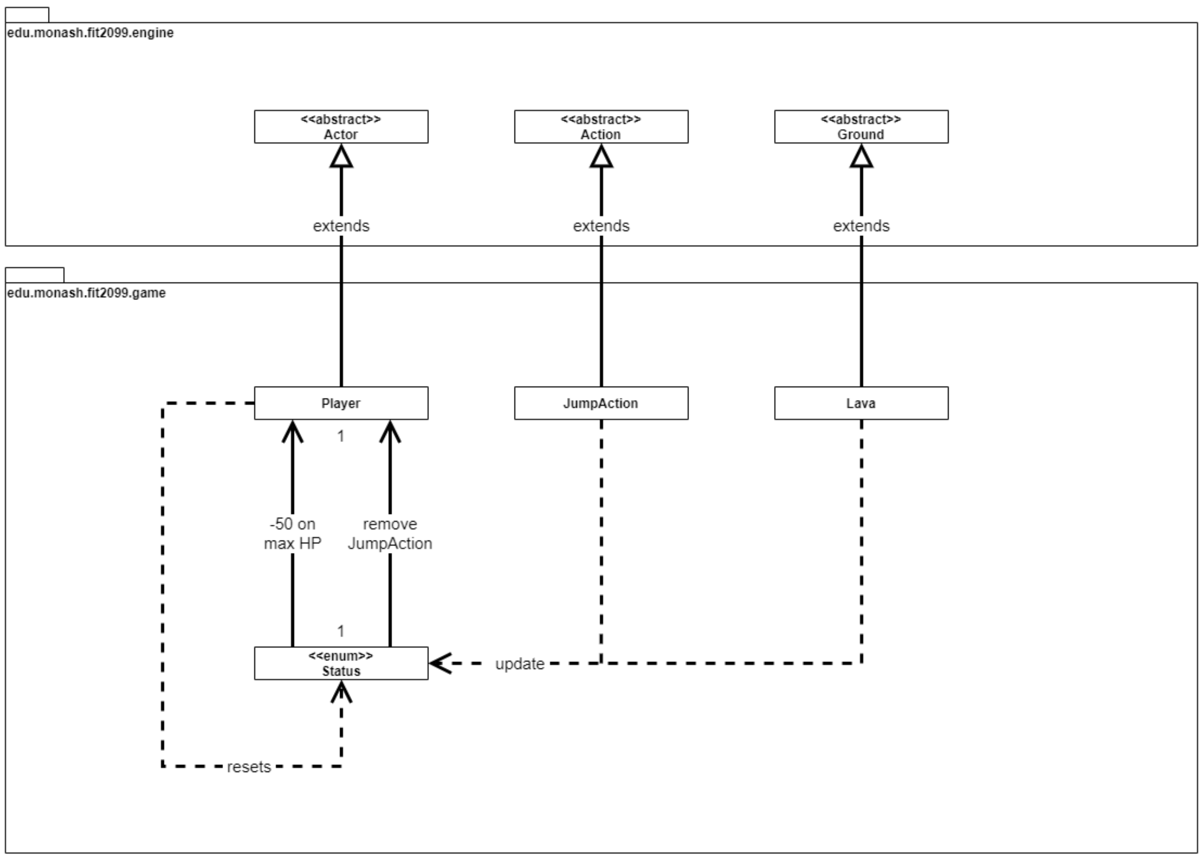


Figure 5 UML diagram describing the involved classes in implementing requirement 4.

2.0 Interaction Diagrams

This part of the document will showcase the interaction diagrams for some of the complex methods in each requirement which involve at least three different classes. Any changes made to the previous implementation in Assignment 2 will be highlighted, followed by a brief explanation of what has been changed and the rationale behind the changes.

2.1 Teleport Action

Note: The image may appear small due to the constraints of the document layout. Clarity, and readability of the image may be improved by zooming in.

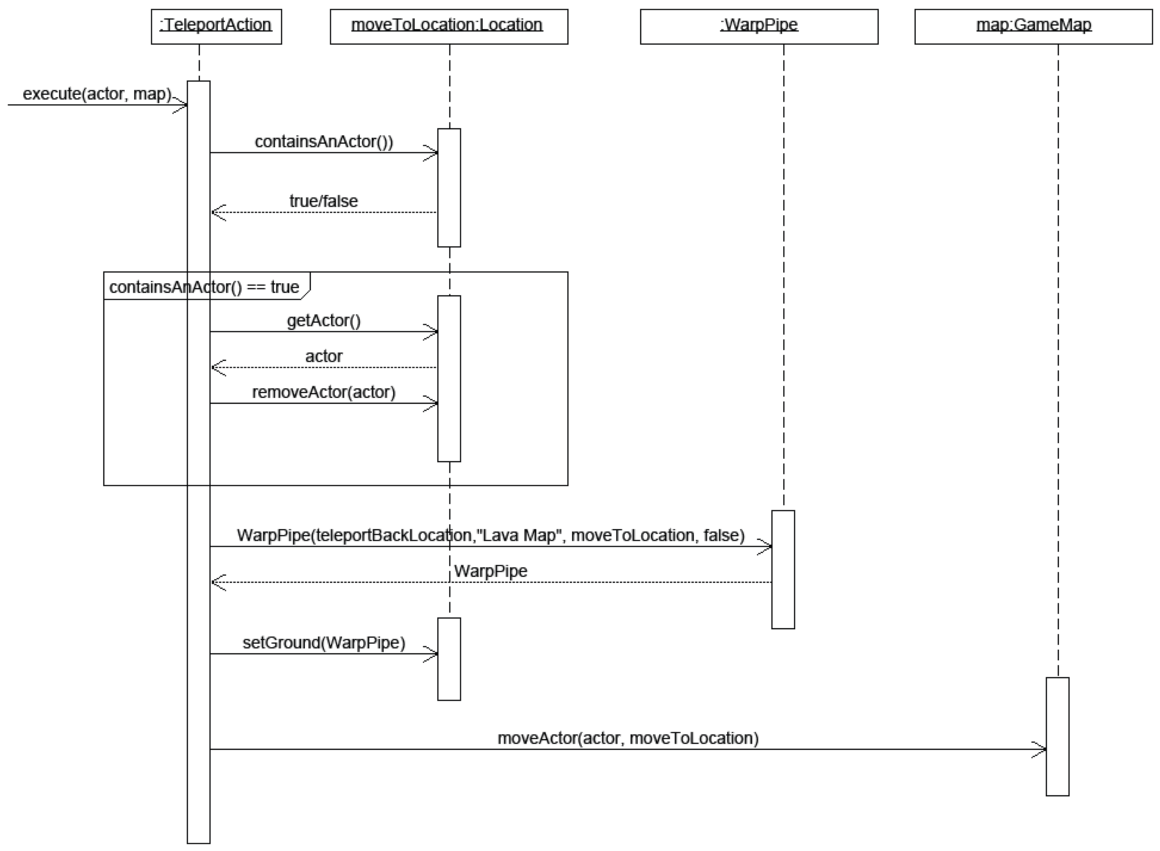


Figure 6 Interaction diagram depicting the interactions between involved classes when a teleport action is executed.

2.2 Filling Action

Note: The image may appear small due to the constraints of the document layout. Clarity, and readability of the image may be improved by zooming in.

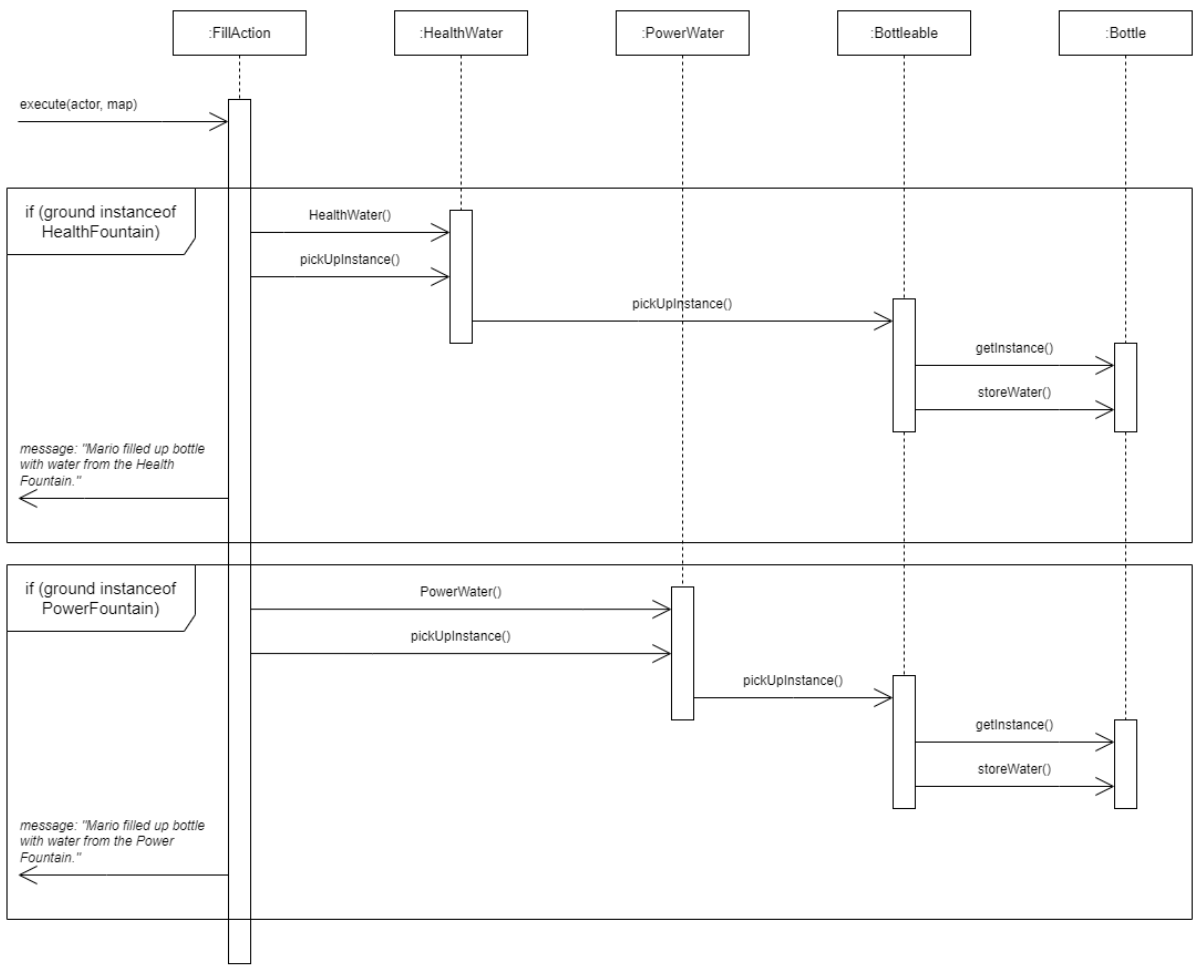
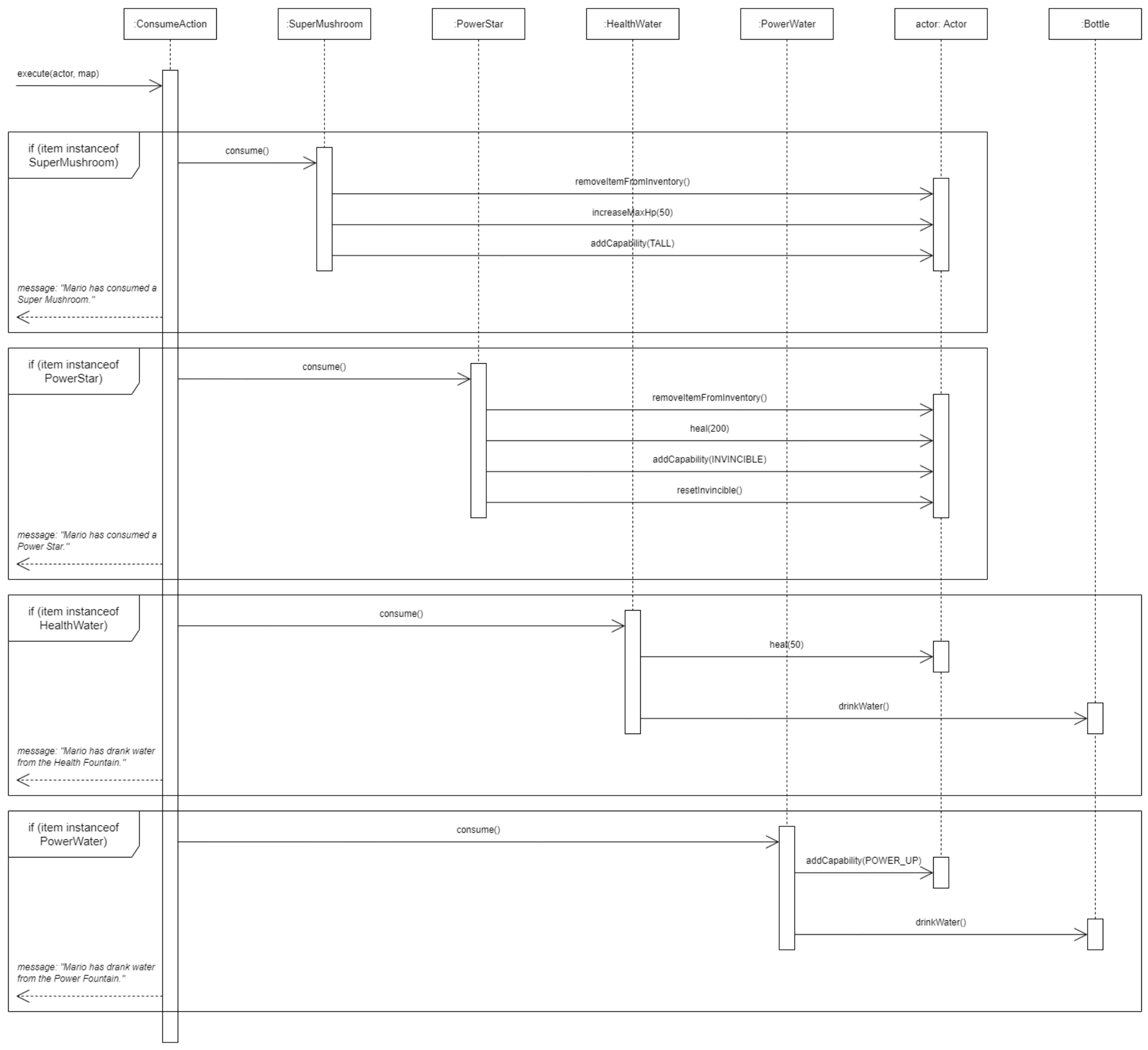

:FillAction
:HealthWater
:PowerWater
:Bottleable
:Bottle
execute(actor, map)
if (ground instanceof HealthFountain)
HealthWater()
pickUpInstance()
pickUpInstance()
getInstance()
storeWater()
message: "Mario filled up bottle with water from the Health Fountain."
if (ground instanceof PowerFountain)
PowerWater()
pickUpInstance()
pickUpInstance()
getInstance()
storeWater()
message: "Mario filled up bottle with water from the Power Fountain."

Figure 7 Interaction diagram depicting the interactions between involved classes when a fill action is executed.

2.3 Consume Action

Note: The image may appear small due to the constraints of the document layout. Clarity, and readability of the image may be improved by zooming in.


:ConsumeAction
:SuperMushroom
:PowerStar
:HealthWater
:PowerWater
actor: Actor
:Bottle
execute(actor, map)
if (item instanceof SuperMushroom)
consume()
removeItemFromInventory()
increaseMaxHp(50)
addCapability(TALL)
message: "Mario has consumed a Super Mushroom."
if (item instanceof PowerStar)
consume()
removeItemFromInventory()
heal(200)
addCapability(INVINCIBLE)
resetInvincible()
message: "Mario has consumed a Power Star."
if (item instanceof HealthWater)
consume()
heal(50)
drinkWater()
message: "Mario has drank water from the Health Fountain."
if (item instanceof PowerWater)
consume()
addCapability(POWER_UP)
drinkWater()
message: "Mario has drank water from the Power Fountain."

3.0 Design Rationale

This part of the document explores the reasoning behind why and how each requirement are tackled with object-oriented programming and the SOLID principles in mind. Some parts of the code may not be discussed if it is of less importance, or it has been provided by the teaching team as resources for the assignment.

3.1 Lava zone

Here following the **single responsibility principle** two different classes have been created for the lava ground and the warp Pipe ground which both extend the ground Class as each of these have different responsibilities such as the lava map Needing to reduce the health points of the player by 15 and warp pipe needing to return jump and teleport actions at different instances

The **open close principle** states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. Here the Lava and Warp pipe have both extended the ground classes. Therefore when creating a new map we must first create a new fancyGroundFactory object which takes in a list of ground objects

```
 FancyGroundFactory lavaZoneGroundFactory = new FancyGroundFactory(new Dirt(),
new Wall(), new Sprout());
```

Here we can extend this lavaZoneGroundFactory to create lava grounds by simply passing in a new Lava Object as per the example below

```
 FancyGroundFactory lavaZoneGroundFactory = new FancyGroundFactory(new Dirt(),
new Wall(), new Lava(), new Sprout());
```

Therefore it is clear that we have extended the this entity but have not made any modifications to the functionality of its class in order to accept the new object.

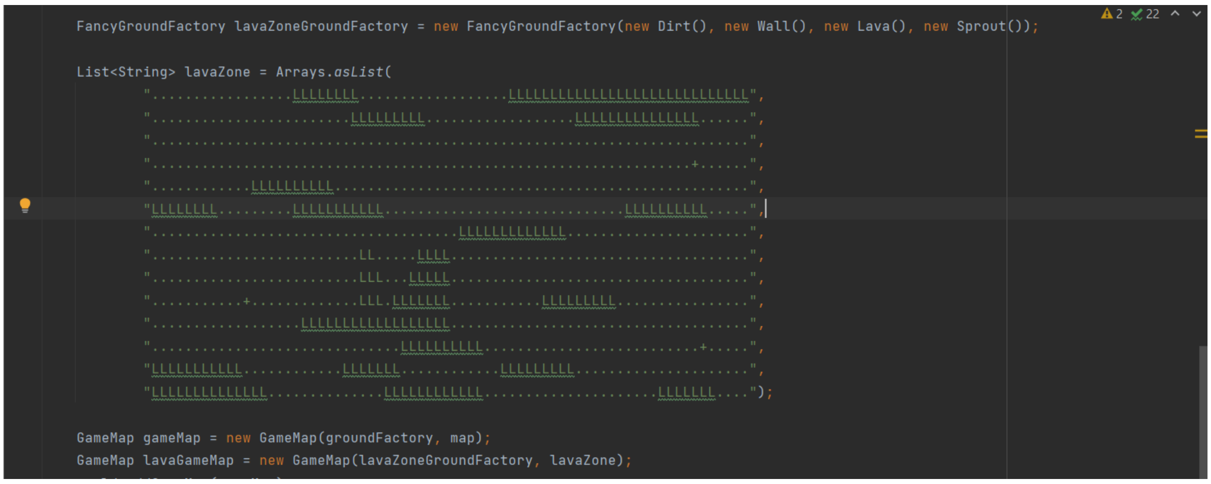
As both the lava and warp pipe extend the Ground class. Even if they do have different responsibilities Non of these two classes are forced to implement functions that are not required for that specific object. Even when considering the Teleport Action which extends the Action it too does not have issues such that it needs to implement functions that the teleport action has no use of. Therefore it is clear that we have not violated the **Interface Segregation Principle.**

Finally it is also clear that both the Lava and Warp classes do not implement any new functions instead each of them just override the functions in the Ground super class and since each and every function has the same signature replacing one with the other should not create any unknown behavior therefore we have followed the **Liskov's Substitution Principle**

Implementation of Features

1. New Map

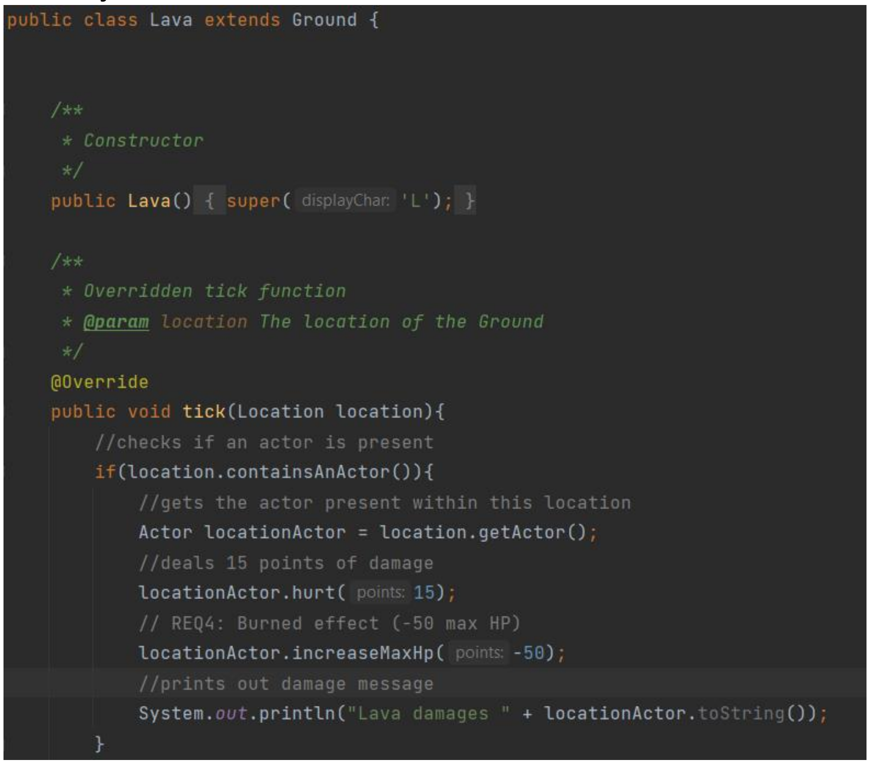
Creating a new Map With Lava Objects that deal 15 damage



```java
FancyGroundFactory lavaZoneGroundFactory = new FancyGroundFactory(new Dirt(), new Wall(), new Lava(), new Sprout());

List<String> lavaZone = Arrays.asList(
        "................LLLLLLLL..................LLLLLLLLLLLLLLLLLLLLLLLLLLLLL",
        ".......................LLLLLLLLL..................LLLLLLLLLLLLLLL......",
        ".......................................................................",
        "................................................................+......",
        "...........LLLLLLLLLL..................................................",
        "LLLLLLLL........LLLLLLLLLLL.............................LLLLLLLLLL.....",
        "....................................LLLLLLLLLLLLL......................",
        "........................LL.....LLLL....................................",
        "........................LLL...LLLLL....................................",
        "...........+............LLL.LLLLLLL...........LLLLLLLLL................",
        ".................LLLLLLLLLLLLLLLLLL....................................",
        ".............................LLLLLLLLLL..........................+.....",
        "LLLLLLLLLLL...........LLLLLLL............LLLLLLLLL.....................",
        "LLLLLLLLLLLLLL.............LLLLLLLLLLLL.....................LLLLLLL....");

GameMap gameMap = new GameMap(groundFactory, map);
GameMap lavaGameMap = new GameMap(lavaZoneGroundFactory, lavaZone);
```

Here a new game map called lavaGameMap has been created. This map is smaller than the main game as requested and filled with Lava Grounds which inflict 15 damage points for any actor that is in its location. The creation of this is done by first creating a groundFactory called lavaZoneGroundFactory which takes in parameters such as Dirt Sprout and Lava Objects. Then I create a list of strings which represents the lavaMap. Then finally these two will be passed on as a parameter to create the gameMap.

Lava Object



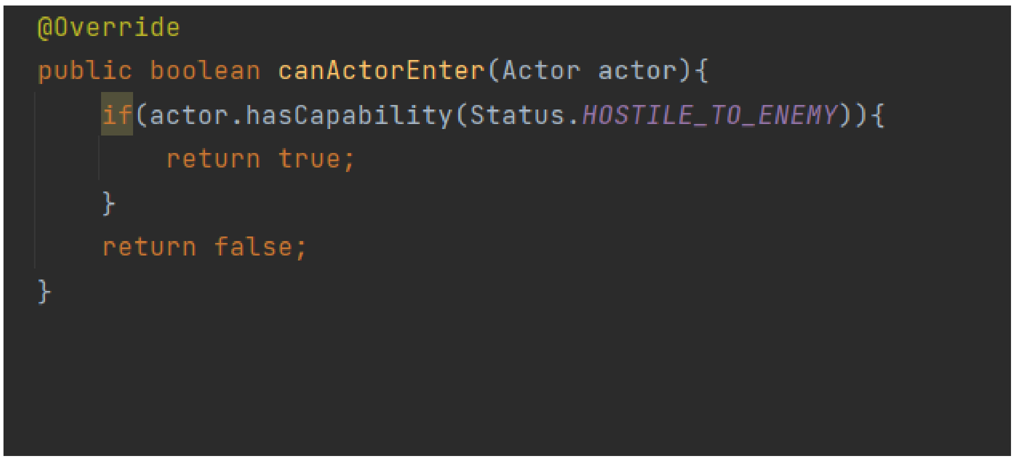
```
public class Lava extends Ground {


    /**
     * Constructor
     */
    public Lava() { super( displayChar: 'L'); }

    /**
     * Overridden tick function
     * @param location The location of the Ground
     */
    @Override
    public void tick(Location location){
        //checks if an actor is present
        if(location.containsAnActor()){
            //gets the actor present within this location
            Actor locationActor = location.getActor();
            //deals 15 points of damage
            locationActor.hurt( points: 15);
            // REQ4: Burned effect (-50 max HP)
            locationActor.increaseMaxHp( points: -50);
            //prints out damage message
            System.out.println("Lava damages " + locationActor.toString());
        }
```

The lava object extends the ground class that is provided.

Dealing 15 Damage

In order to deal 15 damage it will override the tick method of the ground super class. Then I will take the location parameter that is passed within the tick(Location location) method. Which indicates the current ground that the lava is placed in then it will use this location object and check if an Actor is present within this location. If the result is true it will fetch the actor through the getActor() method and then reduce the health points by 15 points using the Actors hurt() function. And then finally print out a message into the console which will indicate that the specific actor has been hurt.

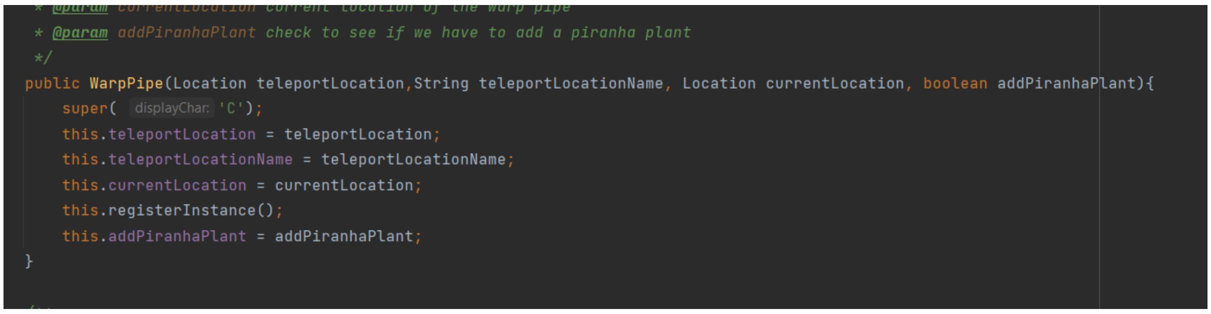
Stopping enemies from stepping into Lava Ground



```
@Override
public boolean canActorEnter(Actor actor){
    if(actor.hasCapability(Status.HOSTILE_TO_ENEMY)){
        return true;
    }
    return false;
}
```

As the player is the only Actor which has the HOSTILE_TO_ENEMY capability. We will use this to check the actor passed in is the player. If it is the player we will return true else it will return false.

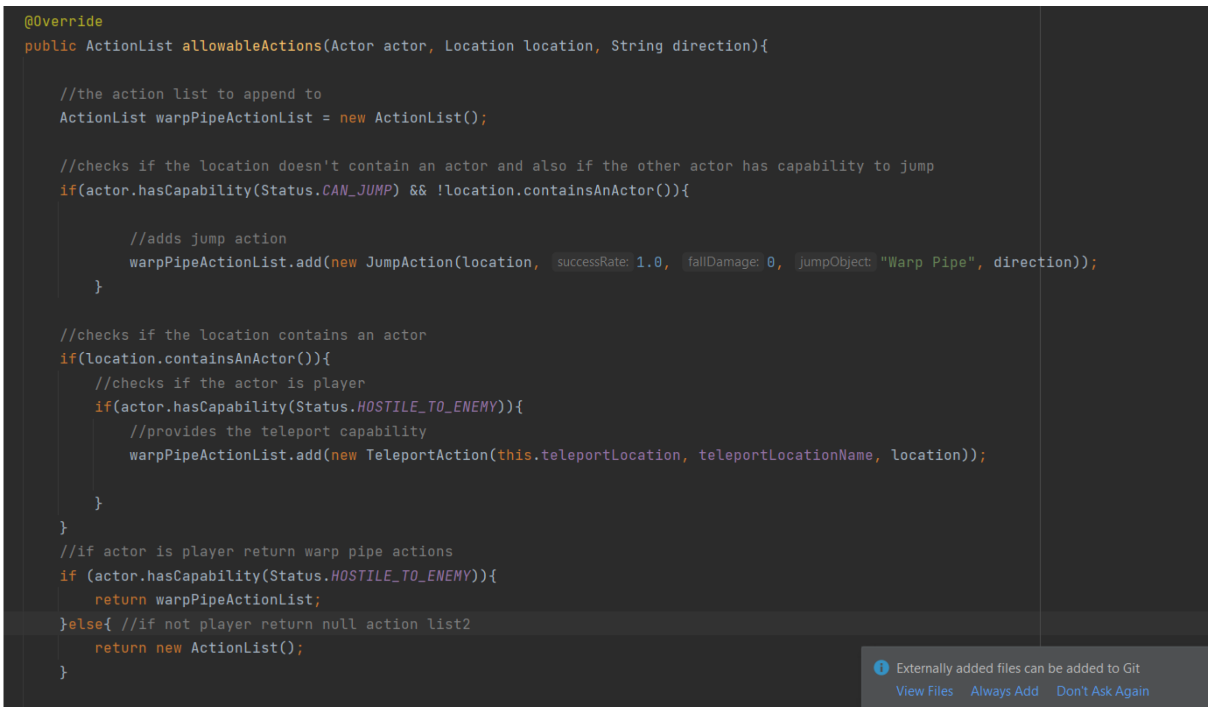
2. Teleportation (Warp Pipe)



```
 * @param currentLocation current location of the warp pipe
 * @param addPiranhaPlant check to see if we have to add a piranha plant
 */
public WarpPipe(Location teleportLocation,String teleportLocationName, Location currentLocation, boolean addPiranhaPlant){
    super( displayChar: 'C');
    this.teleportLocation = teleportLocation;
    this.teleportLocationName = teleportLocationName;
    this.currentLocation = currentLocation;
    this.registerInstance();
    this.addPiranhaPlant = addPiranhaPlant;
}
```

Here a warp Pipe class has been created which extends the ground object. In the constructor of this object it takes in the teleportLocation, name of the location that we are supposed to teleport to, the current location that the warp pipe is currently placed in and a Boolean which will be used to indicate weather or not a piranha plant must be placed within the location of the warp pipe.

Jump Into Warp Pipe

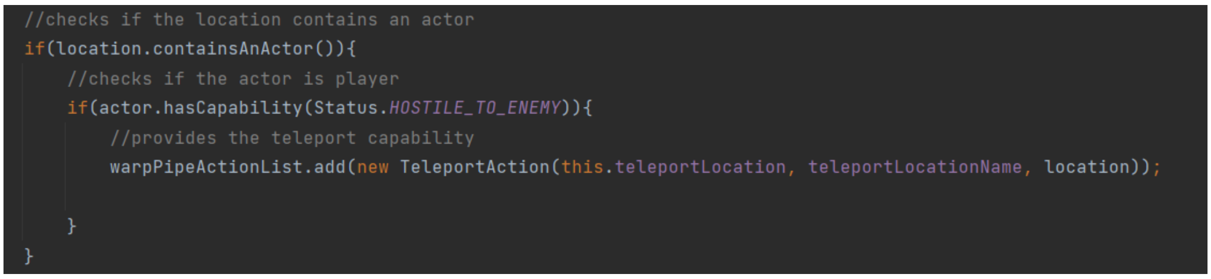


```
@Override
public ActionList allowableActions(Actor actor, Location location, String direction){

    //the action list to append to
    ActionList warpPipeActionList = new ActionList();

    //checks if the location doesn't contain an actor and also if the other actor has capability to jump
    if(actor.hasCapability(Status.CAN_JUMP) && !location.containsAnActor()){

            //adds jump action
            warpPipeActionList.add(new JumpAction(location,  successRate: 1.0,  fallDamage: 0,  jumpObject: "Warp Pipe", direction));
        }

    //checks if the location contains an actor
    if(location.containsAnActor()){
        //checks if the actor is player
        if(actor.hasCapability(Status.HOSTILE_TO_ENEMY)){
            //provides the teleport capability
            warpPipeActionList.add(new TeleportAction(this.teleportLocation, teleportLocationName, location));

        }
    }
    //if actor is player return warp pipe actions
    if (actor.hasCapability(Status.HOSTILE_TO_ENEMY)){
        return warpPipeActionList;
    }else{ //if not player return null action list2
        return new ActionList();
    }
```

Externally added files can be added to Git
View Files   Always Add   Don't Ask Again

Here the player must be able to jump into the location of the warp pipe. Therefore, the allowableActions() method is overridden. Using the actor parameter it checks whether the actor that is requesting the allowable action on the warp pipe contains the capability to jump and also checks that the present location does not contain an actor within it. If both these conditions are satisfied it creates a new jump action and appends it to the action list that is to be returned.

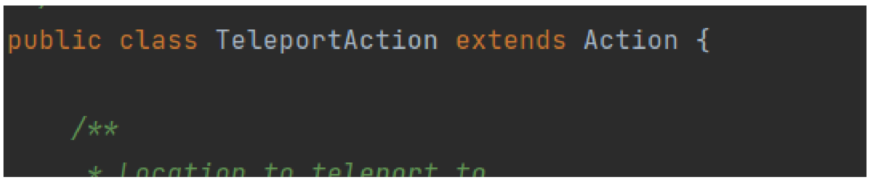
Returning a teleportation Action



```java
//checks if the location contains an actor
if(location.containsAnActor()){
    //checks if the actor is player
    if(actor.hasCapability(Status.HOSTILE_TO_ENEMY)){
        //provides the teleport capability
        warpPipeActionList.add(new TeleportAction(this.teleportLocation, teleportLocationName, location));

    }
}
```

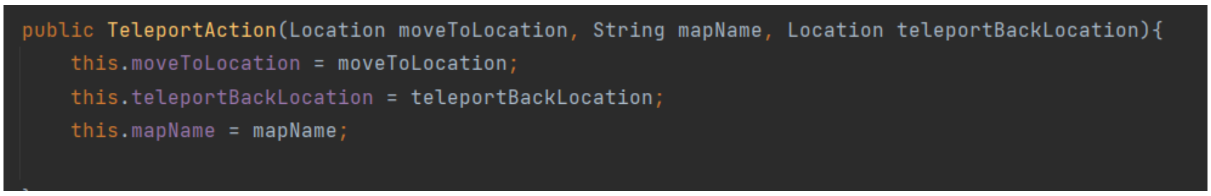
Here in order to return the teleportation action within the allowableAcions() method we first check if the location of the warp pipe contains an actor. We then check if this actor is the player by verifying if the actor contains the HOSTILE_TO_ENEMY capability is present to only the player. If it is confirmed to be the player, we return the Teleport Action

The Teleport Action



```java
public class TeleportAction extends Action {


    /**
     * Location to teleport to
```

The teleport action is subclass which extends the action class.



```java
public TeleportAction(Location moveToLocation, String mapName, Location teleportBackLocation){
    this.moveToLocation = moveToLocation;
    this.teleportBackLocation = teleportBackLocation;
    this.mapName = mapName;

```

The constructor of this class takes in the location to move to, name of the location to move to and the teleportBackLocation.

Teleporting to the necessary location and teleporting back to the same location

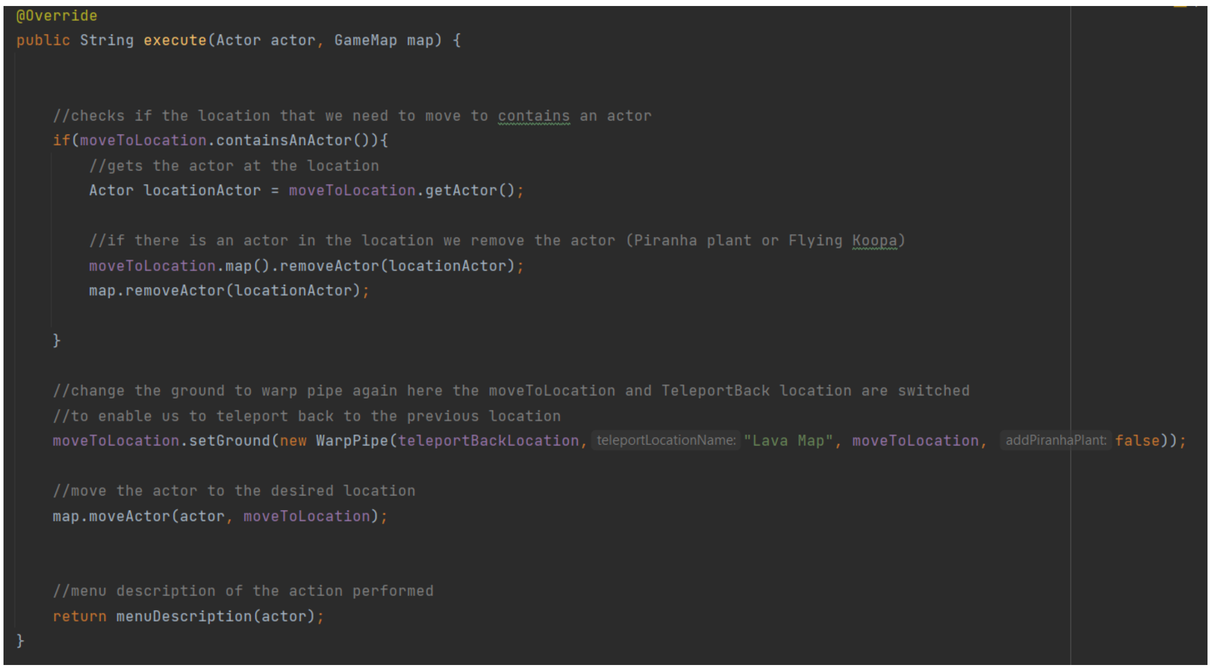
Here when executing this action we first check whether the location that we want to moveToLocation contains an actor. If this is true we then remove that actor.

Before we move the actor to the desired location we first run this command below

moveToLocation.setGround(new WarpPipe(teleportBackLocation,"Lava Map", moveToLocation, false));

this command sets the ground of the teleport location to a new warp pipe which doesn't spawn piranha plant. Another important note is that in this the moveToLocation and TeleportBackLOcation are switched in this Warp Pipe. This is done so as to allow the actor to teleport back to the exact previous location(the warp pipe) which teleported the player to his current location.

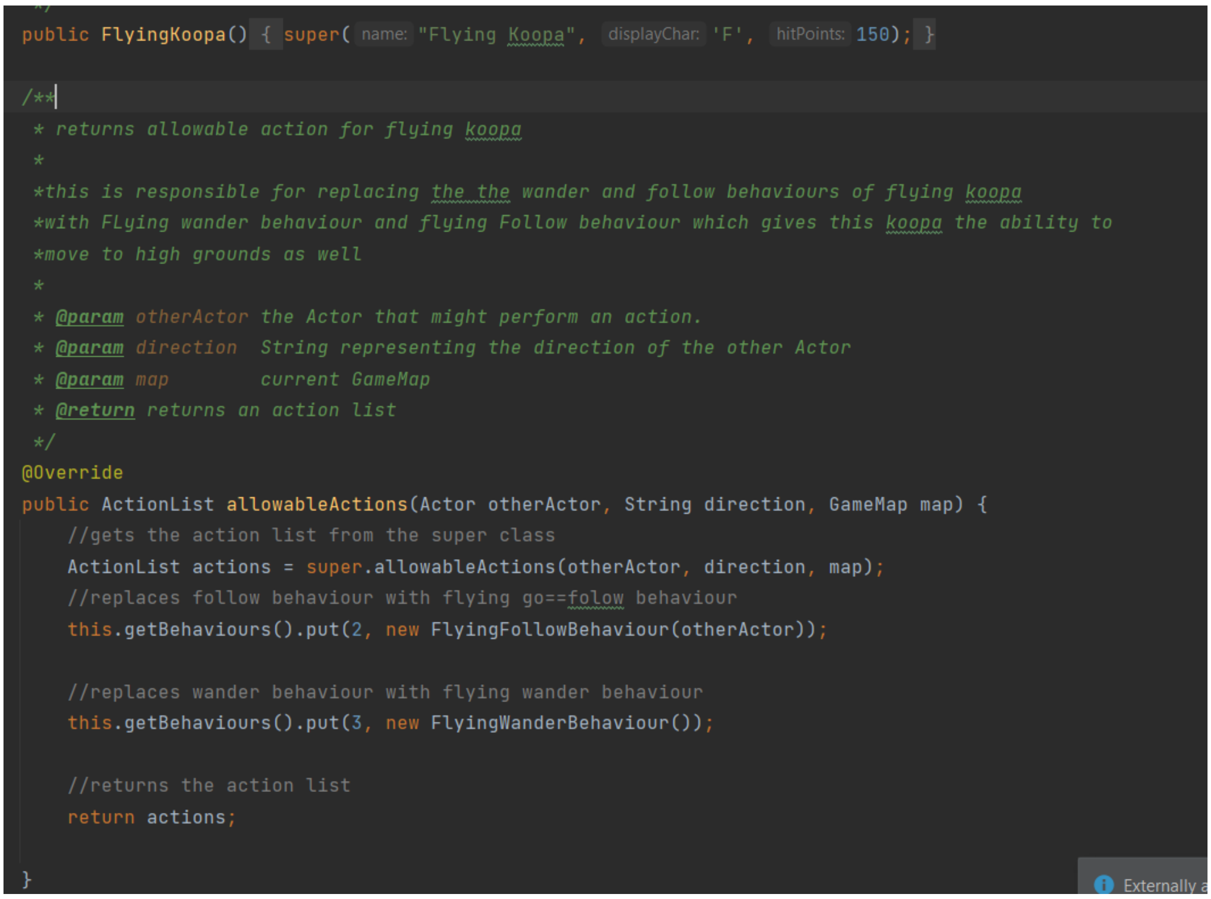
Finally we then move the player to the moveToLocation using the map parameter passed into the execute() method.



```java
@Override
public String execute(Actor actor, GameMap map) {


    //checks if the location that we need to move to contains an actor
    if(moveToLocation.containsAnActor()){
        //gets the actor at the location
        Actor locationActor = moveToLocation.getActor();

        //if there is an actor in the location we remove the actor (Piranha plant or Flying Koopa)
        moveToLocation.map().removeActor(locationActor);
        map.removeActor(locationActor);

    }

    //change the ground to warp pipe again here the moveToLocation and TeleportBack location are switched
    //to enable us to teleport back to the previous location
    moveToLocation.setGround(new WarpPipe(teleportBackLocation, teleportLocationName: "Lava Map", moveToLocation,  addPiranhaPlant: false));

    //move the actor to the desired location
    map.moveActor(actor, moveToLocation);


    //menu description of the action performed
    return menuDescription(actor);
}
```

3.2 More allies and enemies!

Here in fulling these requirements it is clear that I have followed the dry principle. This can be verified by taking a look into the FlyingKoopa Class which extends the Koopa Class. Since most of them share mostly the same functionality only the allowable actions class has been made a slight change to replace the Wander and Follow Behaviors with the Flying Wander Behavior and the Flying Follow Behavior.



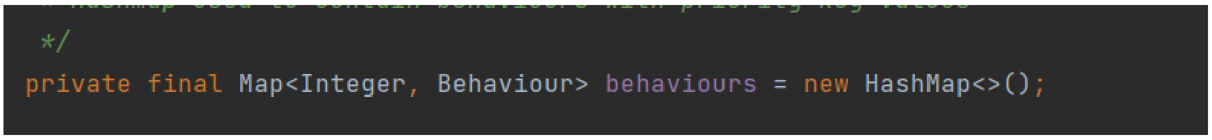
```
public FlyingKoopa() { super( name: "Flying Koopa", displayChar: 'F', hitPoints: 150); }

/**
 * returns allowable action for flying koopa
 *
 *this is responsible for replacing the the wander and follow behaviours of flying koopa
 *with FLying wander behaviour and flying Follow behaviour which gives this koopa the ability to
 *move to high grounds as well
 *
 * @param otherActor the Actor that might perform an action.
 * @param direction  String representing the direction of the other Actor
 * @param map        current GameMap
 * @return returns an action list
 */
@Override
public ActionList allowableActions(Actor otherActor, String direction, GameMap map) {
    //gets the action list from the super class
    ActionList actions = super.allowableActions(otherActor, direction, map);
    //replaces follow behaviour with flying go==folow behaviour
    this.getBehaviours().put(2, new FlyingFollowBehaviour(otherActor));

    //replaces wander behaviour with flying wander behaviour
    this.getBehaviours().put(3, new FlyingWanderBehaviour());

    //returns the action list
    return actions;

}
```

Therefore it is clear that I have not repeated any unnecessary code when creating this class.

Separate classes have been created for the Bowser, Peach and Flying Koopa classes as each of them have different types of responsibilities and actions to return. Such as peach not having any behaviors implemented and is required to return the end game action when the player has the required key and Bowser requiring to implement different Attack Fire Methods. Therefore, as each of these unique responsibilities are being separated to different classes it is clear that the **Single Responsibility Principle** is being followed

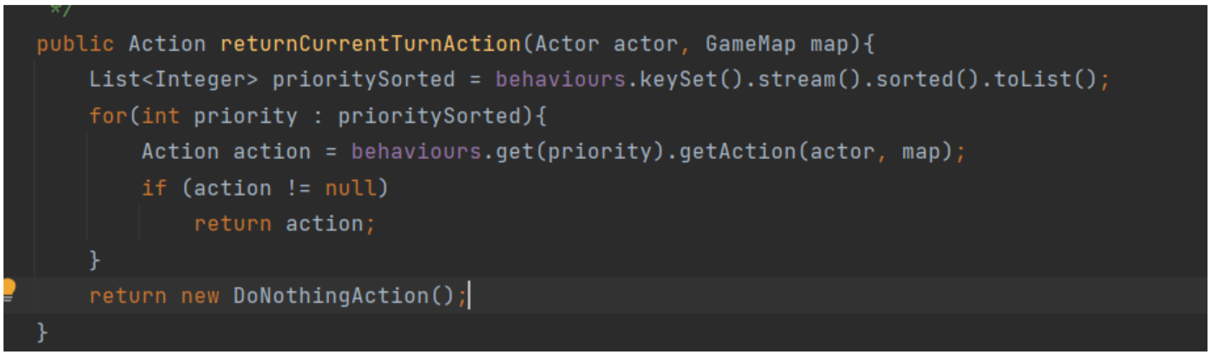
The **Liskov Substitution Principle** states that any subclass object should be substitutable for the superclass object from which it is derived. And since the only difference in Flying Koopa and Koopa is the behaviors which they implement and both having mostly the same functionality we can easily replace one with the other without the worry of any unknow behavior that will cause errors from occurring.

Creating different behaviors for the Flying Koopa class such as the Flying Wander Behavior and the Flying Follow Behavior and for Bowser the Attack Fire Behavior enables us to expand the functionality as we are adding these new behaviours to the hash map that is present within the Enemy Type Subclasses



```java
 */
private final Map<Integer, Behaviour> behaviours = new HashMap<>();
```

And since all these behaviors have the getAction(actor, map) function we can call upon them in the following method without changing its implementation as shown below



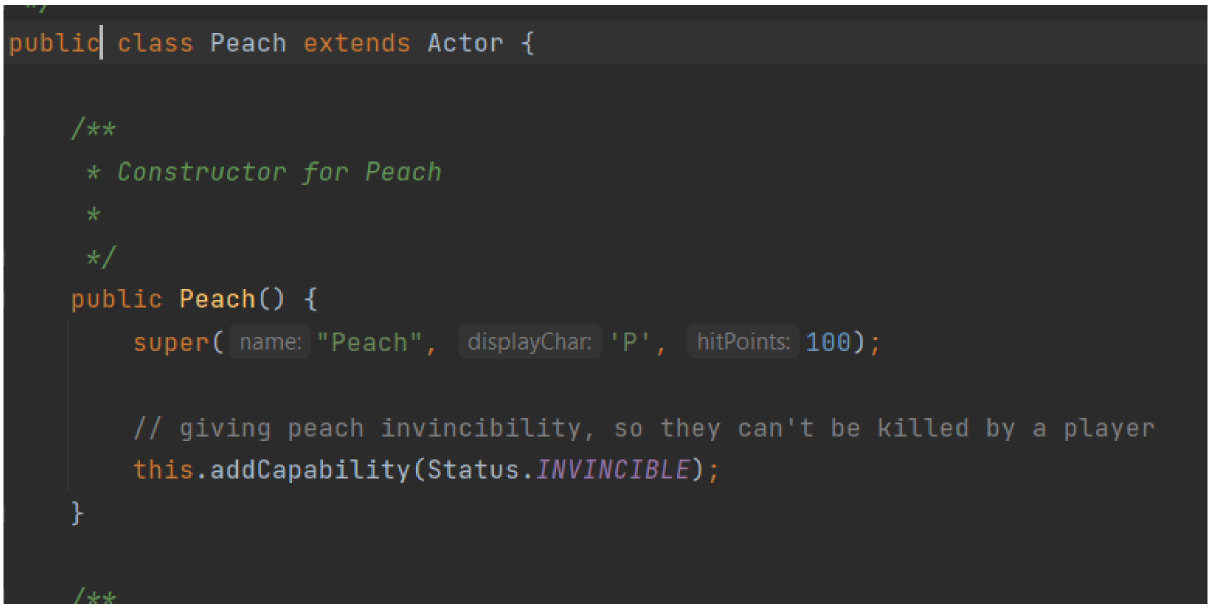
```java
 */
public Action returnCurrentTurnAction(Actor actor, GameMap map){
    List<Integer> prioritySorted = behaviours.keySet().stream().sorted().toList();
    for(int priority : prioritySorted){
        Action action = behaviours.get(priority).getAction(actor, map);
        if (action != null)
            return action;
    }
    return new DoNothingAction();
}
```

Therefore it is clear that we are following the principle of the **Open Close Principle.**

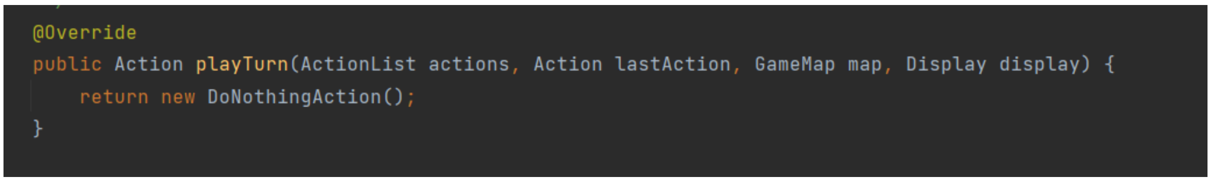
In any of these New enemy type objects or new behaviors we are not in any instance implementing functions which are irrelevant to the specific class. Therefore the **Interface Segregation Principle** is also not violated.

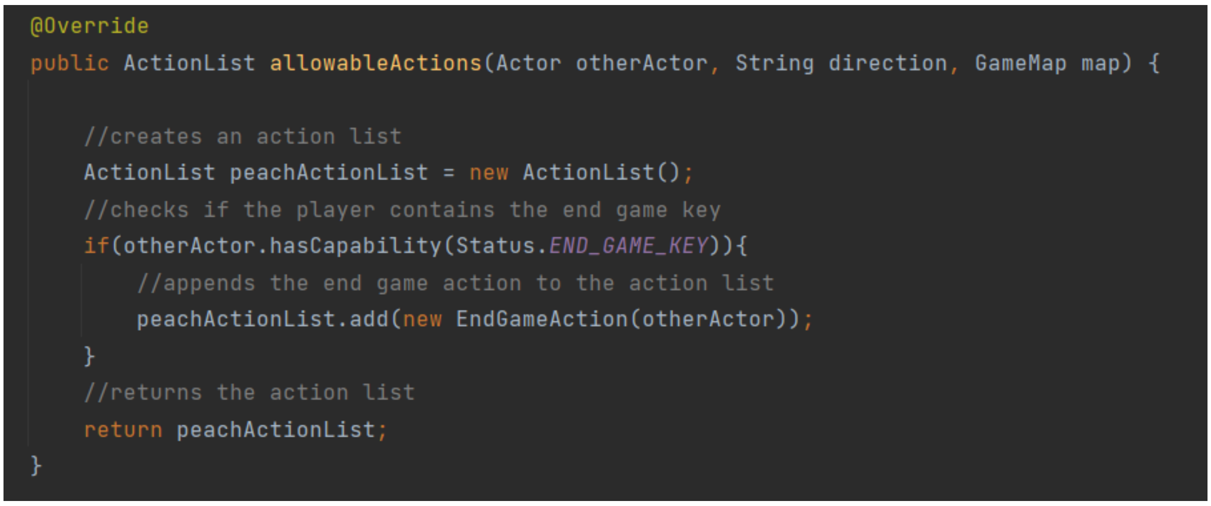
Requirement 2 Implementation

1. **Princess Peach**


```java
public class Peach extends Actor {

    /**
     * Constructor for Peach
     *
     */
    public Peach() {
        super( name: "Peach",  displayChar: 'P',  hitPoints: 100);

        // giving peach invincibility, so they can't be killed by a player
        this.addCapability(Status.INVINCIBLE);
    }

    /**
```

Here the princess peach is a class which extends the actor. In the constructor princess peach is given the INVINCIBLE capability so that she may not be harmed by the player.

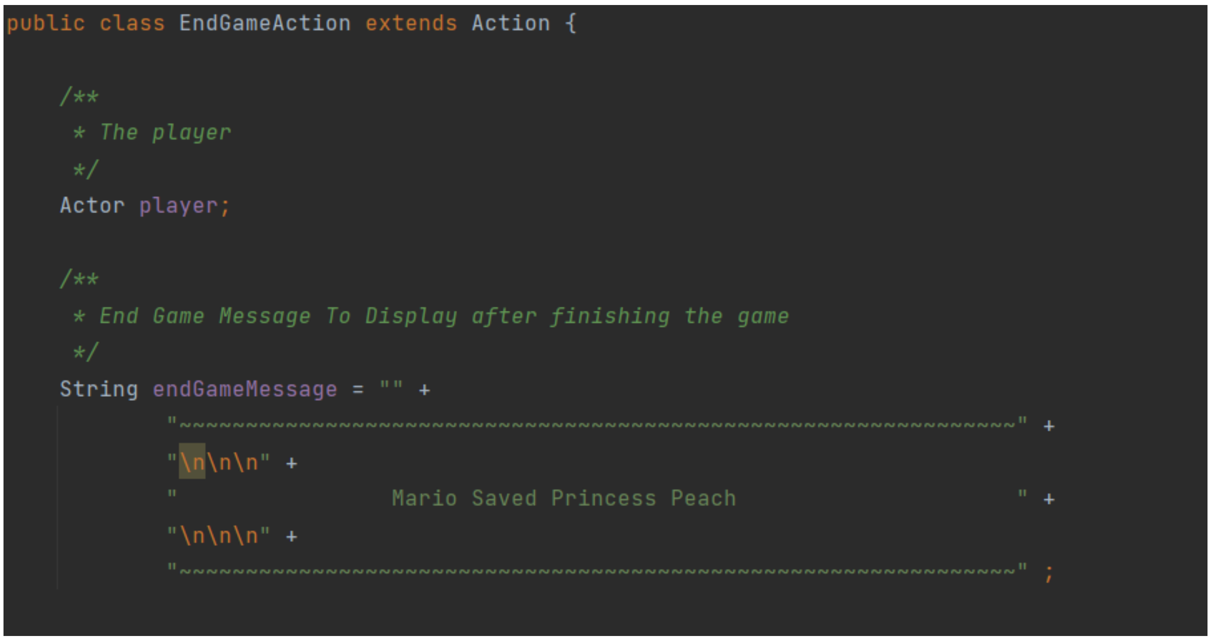

```java
@Override
public Action playTurn(ActionList actions, Action lastAction, GameMap map, Display display) {
    return new DoNothingAction();
}
```


```java
@Override
public ActionList allowableActions(Actor otherActor, String direction, GameMap map) {

    //creates an action list
    ActionList peachActionList = new ActionList();
    //checks if the player contains the end game key
    if(otherActor.hasCapability(Status.END_GAME_KEY)){
        //appends the end game action to the action list
        peachActionList.add(new EndGameAction(otherActor));
    }
    //returns the action list
    return peachActionList;
}
```

Princess Peach always returns a D0NothingAction every turn as she cannot move, attack, or be attacked.In the allowabelActions() method this object first checks if the other actor has the end game key in their inventory. If this is true then a special action EndGameAction which will unlock princess peach's handcuffs and end the game will be returned

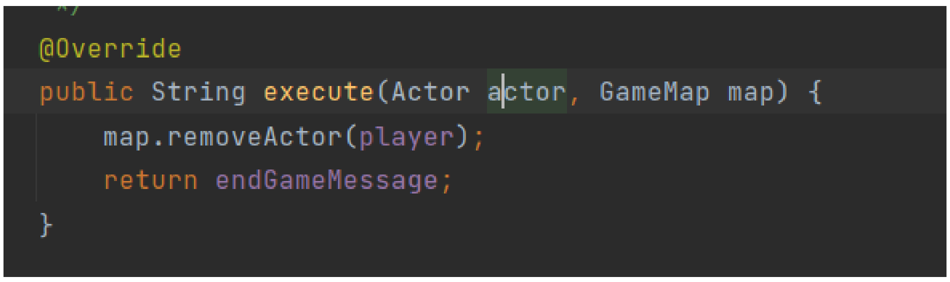
EndGameAction

Here the EndGameAction is a class that extends the action.



```java
public class EndGameAction extends Action {

    /**
     * The player
     */
    Actor player;

    /**
     * End Game Message To Display after finishing the game
     */
    String endGameMessage = "" +
            "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" +
            "\n\n\n" +
            "                Mario Saved Princess Peach                     " +
            "\n\n\n" +
            "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" ;
```

It has the attribute of an actor and the string endGameMessage which will be displayed once the game is over.

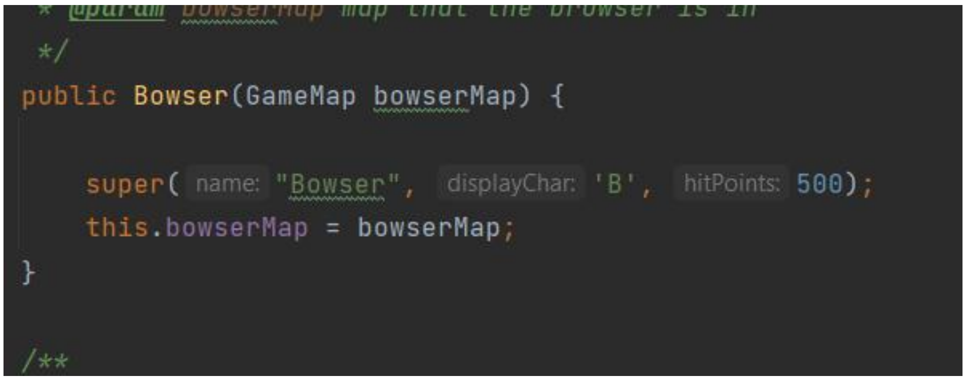


```java
@Override
public String execute(Actor actor, GameMap map) {
    map.removeActor(player);
    return endGameMessage;
}
```

When this method is executed it will remove the player from the map which will effectively end the game. Then it will display the end game message in the console.

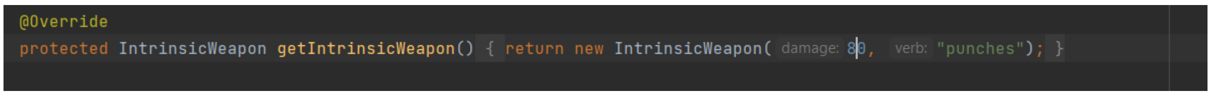
2. Bowser B

Here the Bowser is a class which extends the EnemyType class in it’s constructor it will take in a game map which will be used later to rest its position. This bowser has 500 hit points.



```
 * @param bowserMap map that the browser is in
 */
public Bowser(GameMap bowserMap) {

    super( name: "Bowser",  displayChar: 'B',  hitPoints: 500);
    this.bowserMap = bowserMap;
}

/**
```

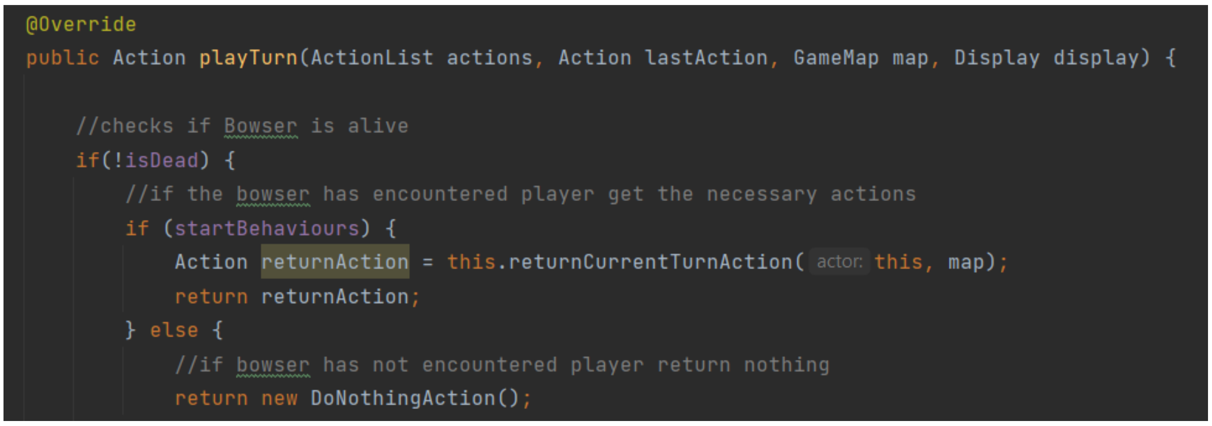
Here the bowser has an intrinsic weapon “punch” which deals 80 damage



```
@Override
protected IntrinsicWeapon getIntrinsicWeapon() { return new IntrinsicWeapon( damage: 80,  verb: "punches"); }
```

In the Bowsers playTurn() method it will first check if the bowser is still alive. If the bowser is alive

And the flag startBehaviours is true(set to true when player is encountered) then the appropriation

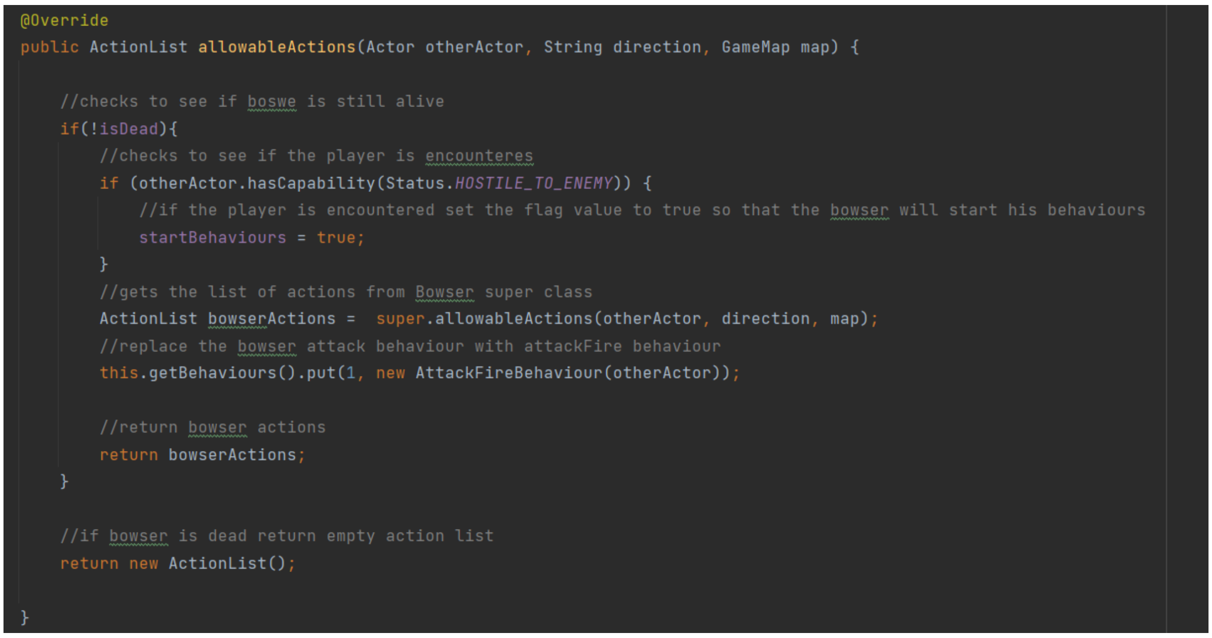


```
@Override
public Action playTurn(ActionList actions, Action lastAction, GameMap map, Display display) {

    //checks if Bowser is alive
    if(!isDead) {
        //if the bowser has encountered player get the necessary actions
        if (startBehaviours) {
            Action returnAction = this.returnCurrentTurnAction( actor: this, map);
            return returnAction;
        } else {
            //if bowser has not encountered player return nothing
            return new DoNothingAction();
```

action will be returned by calling the returnCurrentTurnAction() function from the super Class else it will return a Do Nothing action

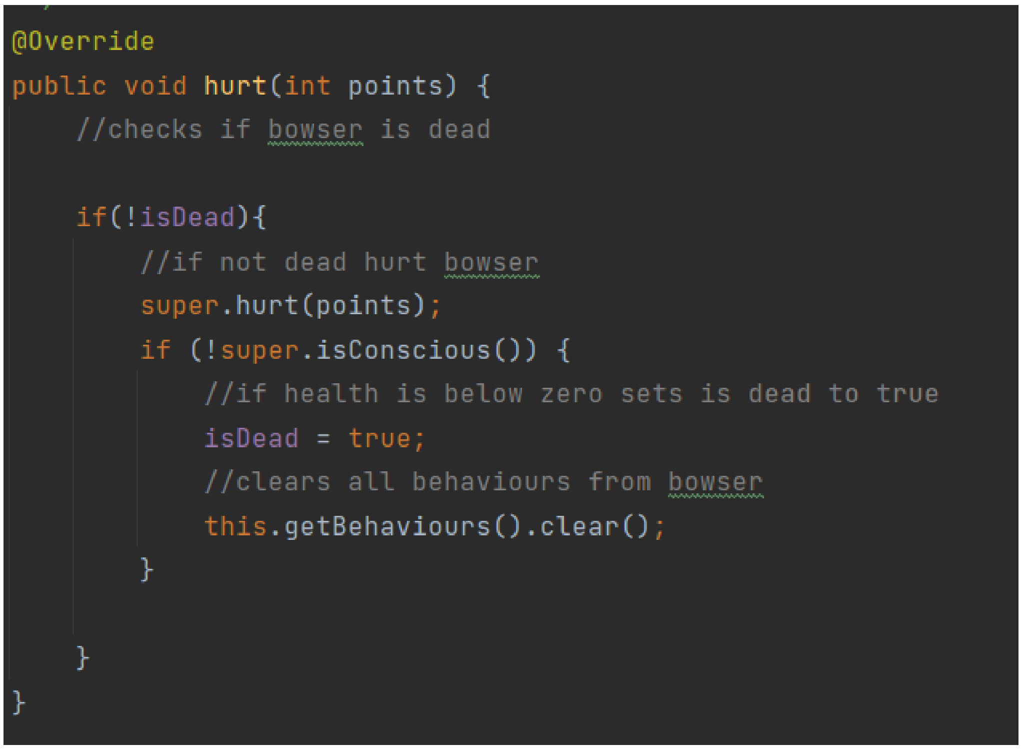
In the allowable actions of bowser it will first check if the bowser is still alive. If it is alive it will then check if the actor requesting the actions has the capability HOSTILE_TO_ENEMY which is present only within the player object. If so it will set the start behaviour to true where Bowser will start implementing its behaviors when its next turn comes   After this will call the allowable actions from the super class and get the action list.

Then we will replace the Attack action behavior from bowser and replace it with AttackFireBehaviour which will spawn Fire when the bowser decides to attack. Finally it will return the action list



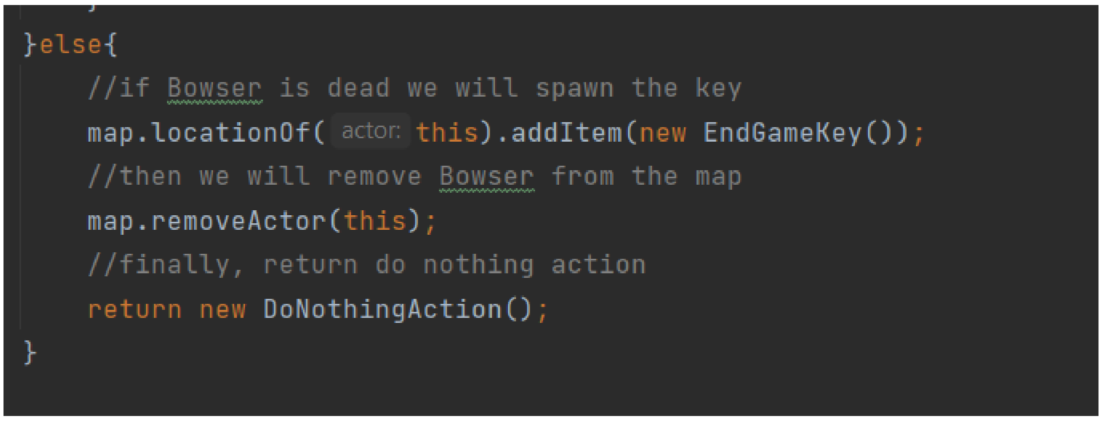
```java
@Override
public ActionList allowableActions(Actor otherActor, String direction, GameMap map) {

    //checks to see if boswe is still alive
    if(!isDead){
        //checks to see if the player is encounteres
        if (otherActor.hasCapability(Status.HOSTILE_TO_ENEMY)) {
            //if the player is encountered set the flag value to true so that the bowser will start his behaviours
            startBehaviours = true;
        }
        //gets the list of actions from Bowser super class
        ActionList bowserActions =  super.allowableActions(otherActor, direction, map);
        //replace the bowser attack behaviour with attackFire behaviour
        this.getBehaviours().put(1, new AttackFireBehaviour(otherActor));

        //return bowser actions
        return bowserActions;
    }

    //if bowser is dead return empty action list
    return new ActionList();

}
```

Whenever the bowser is attacked it will check to see where its health points are less than zero.



```java
@Override
public void hurt(int points) {
    //checks if bowser is dead

    if(!isDead){
        //if not dead hurt bowser
        super.hurt(points);
        if (!super.isConscious()) {
            //if health is below zero sets is dead to true
            isDead = true;
            //clears all behaviours from bowser
            this.getBehaviours().clear();
        }

    }
}
```

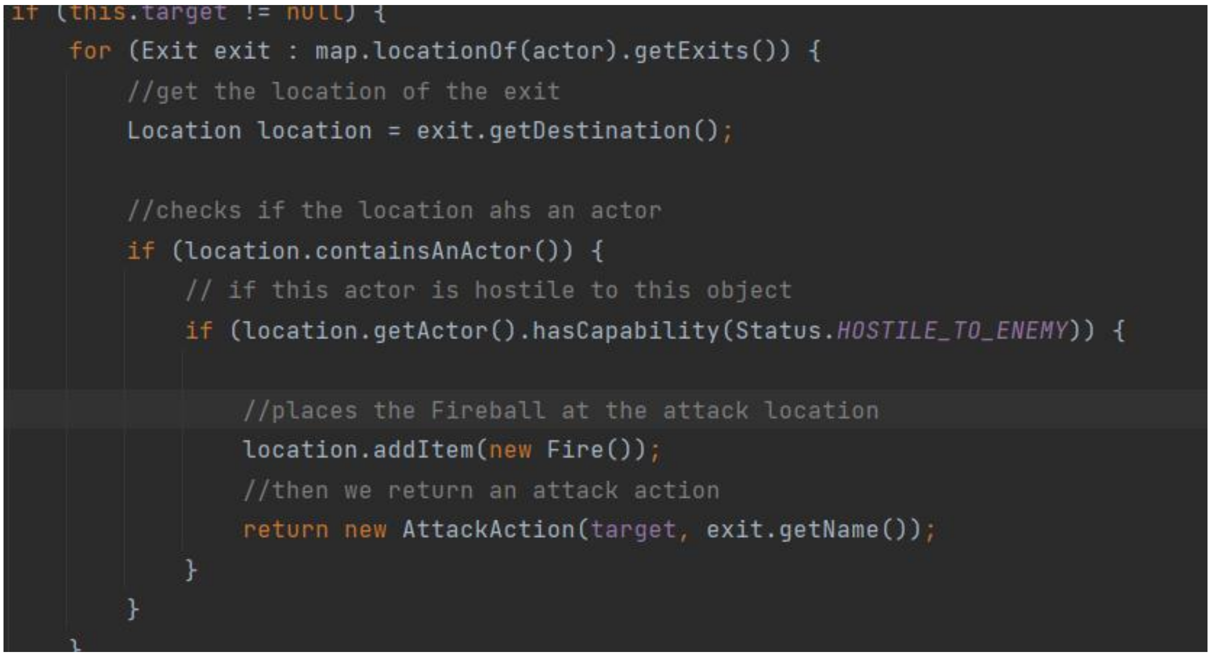
If it is less than zero then we set the isDead Boolean to true and clear all of bowsers behaviors so that it does not do anything. After that in the next bowser turn when the isDead Boolean indicates that bowser is dead the bowser will spawn the EndGameKey in its current location then it will remove itself using the game map provided and a do nothing action will be returned.



```
    }
}else{
    //if Bowser is dead we will spawn the key
    map.locationOf( actor: this).addItem(new EndGameKey());
    //then we will remove Bowser from the map
    map.removeActor(this);
    //finally, return do nothing action
    return new DoNothingAction();
}
```

Attack Fire Behavior

As the Bowser is required to spawn fire each time it attacks

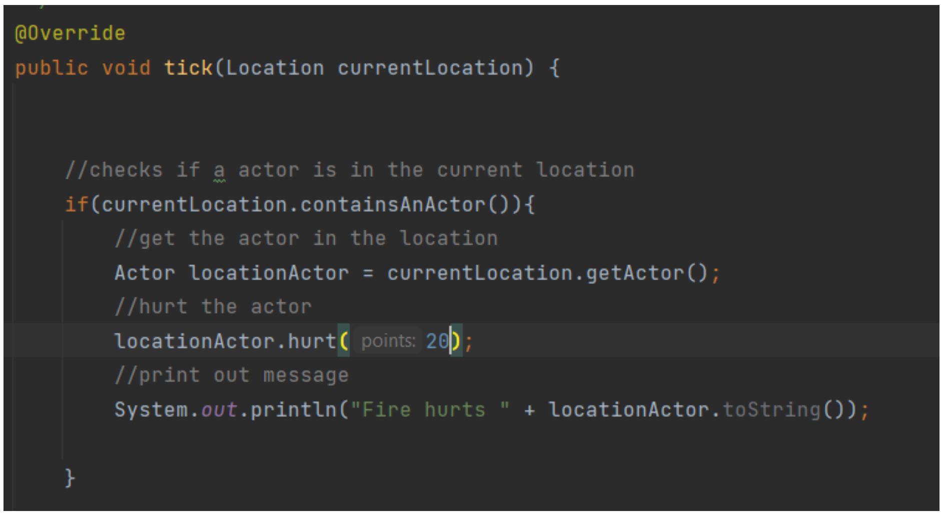


```
if (this.target != null) {
    for (Exit exit : map.locationOf(actor).getExits()) {
        //get the location of the exit
        Location location = exit.getDestination();

        //checks if the location ahs an actor
        if (location.containsAnActor()) {
            // if this actor is hostile to this object
            if (location.getActor().hasCapability(Status.HOSTILE_TO_ENEMY)) {

                //places the Fireball at the attack location
                location.addItem(new Fire());
                //then we return an attack action
                return new AttackAction(target, exit.getName());
            }
        }
    }
```

In order to accomplish this the AttackFireBehaviour acts just like the Attack Behaviour however right before returning the AttackAction which will be executed this behavior will spawn a fire in the location where the target is.

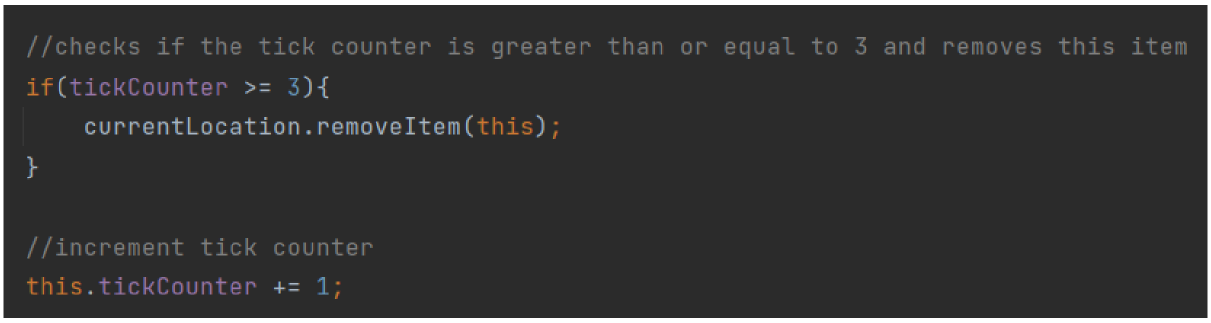
Fire Object.

The fire Object is an object spawned by the bowser attack which will deal 10 damage to anything within its location. This is accomplished in the tick method of the object. It will get the current location and check whether there is an Actor present within the current location. If it is true then the fire will reduce that actors health by 20 points



```java
@Override
public void tick(Location currentLocation) {


    //checks if a actor is in the current location
    if(currentLocation.containsAnActor()){
        //get the actor in the location
        Actor locationActor = currentLocation.getActor();
        //hurt the actor
        locationActor.hurt( points: 20);
        //print out message
        System.out.println("Fire hurts " + locationActor.toString());

    }
```

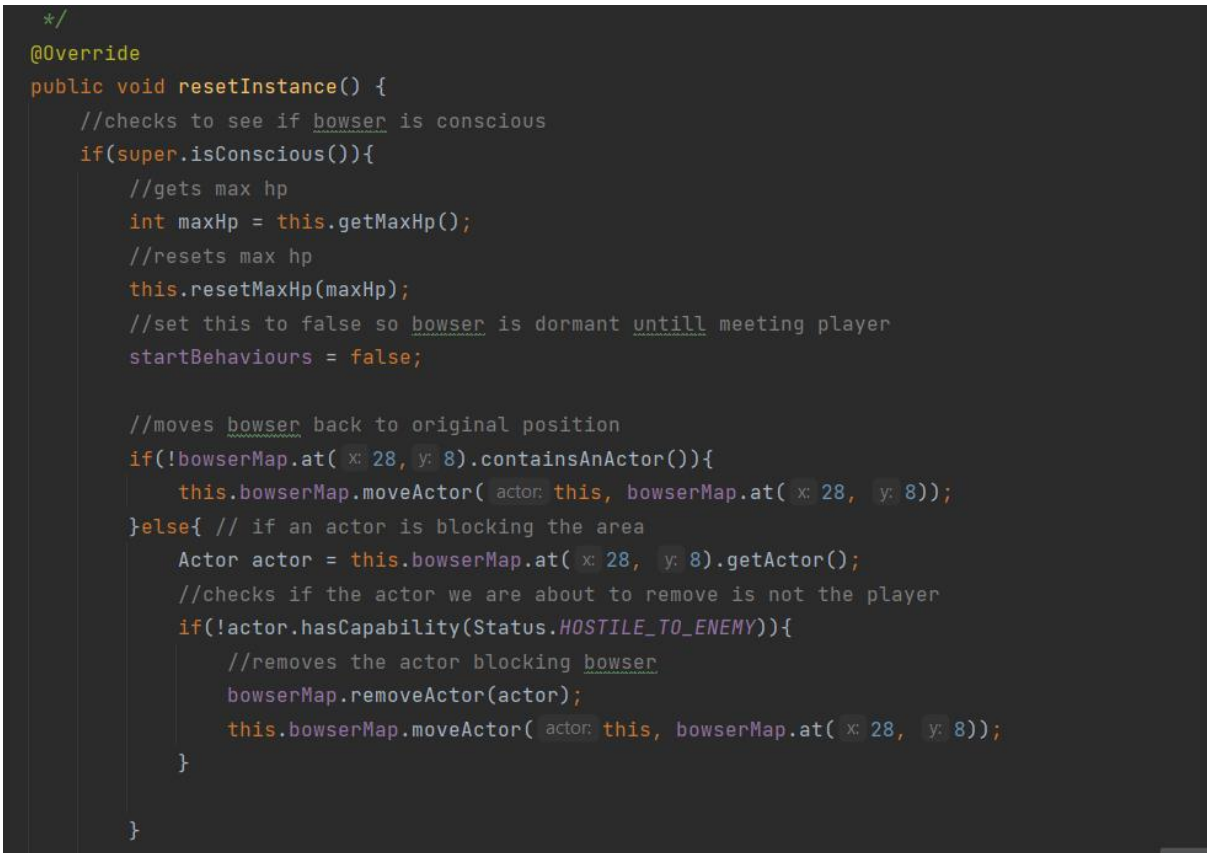
Then finally this fire object should disappear after three turns. This will be done by using a tickCounter which will be incremented at the end of the tick method. Then if this tickCounter value is greater than 3 the item will be removed from the location



```java
//checks if the tick counter is greater than or equal to 3 and removes this item
if(tickCounter >= 3){
    currentLocation.removeItem(this);
}

//increment tick counter
this.tickCounter += 1;
```

Bowser Reset Behavior

The bowser must return to its original position when the reset is performed therefore



```
 */
@Override
public void resetInstance() {
    //checks to see if bowser is conscious
    if(super.isConscious()){
        //gets max hp
        int maxHp = this.getMaxHp();
        //resets max hp
        this.resetMaxHp(maxHp);
        //set this to false so bowser is dormant untill meeting player
        startBehaviours = false;

        //moves bowser back to original position
        if(!bowserMap.at( x: 28, y: 8).containsAnActor()){
            this.bowserMap.moveActor( actor: this, bowserMap.at( x: 28,  y: 8));
        }else{ // if an actor is blocking the area
            Actor actor = this.bowserMap.at( x: 28,  y: 8).getActor();
            //checks if the actor we are about to remove is not the player
            if(!actor.hasCapability(Status.HOSTILE_TO_ENEMY)){
                //removes the actor blocking bowser
                bowserMap.removeActor(actor);
                this.bowserMap.moveActor( actor: this, bowserMap.at( x: 28,  y: 8));
            }

        }
```

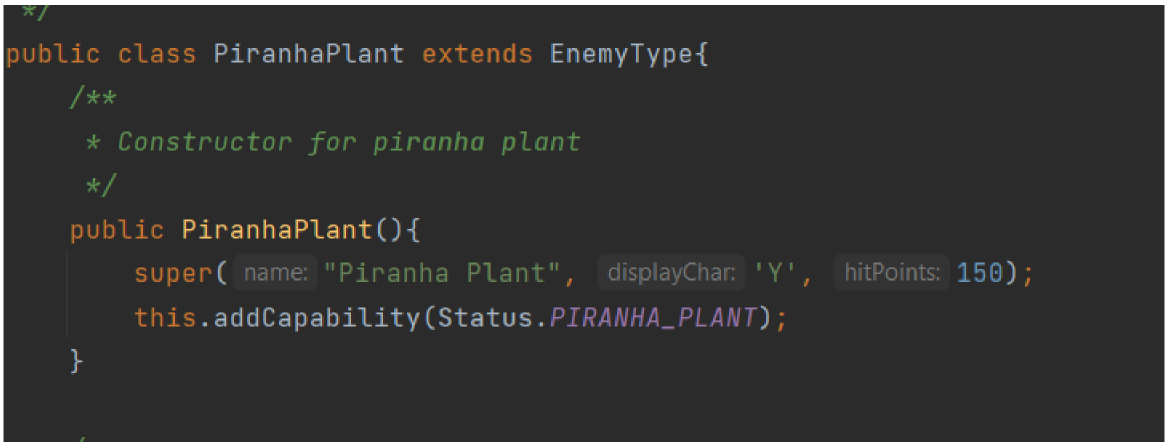
Here to accomplish this in the reset instance of the Bowser it will use its gameMap Attribute.

Firstly it will check if the Bowser is alive using the isConcious() method in the super class. If it is alive then the hp of bowser will be reset using the resetMaxHP() method.

Finally it will check if the location that the bowser is supposed to move to contains an actor if there is an actor it and the actor is not the player it will remove that actor then it will move Bowser to that location.

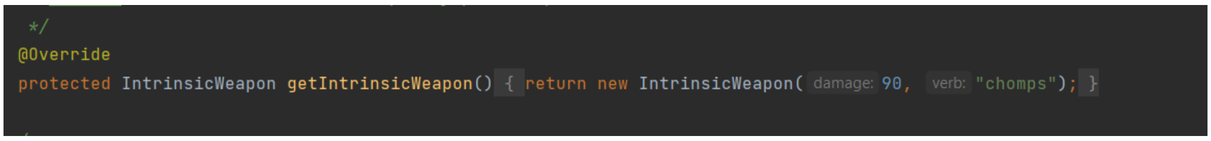
3. Piranha Plant

Here the Piranha Plant extends the Actor Class



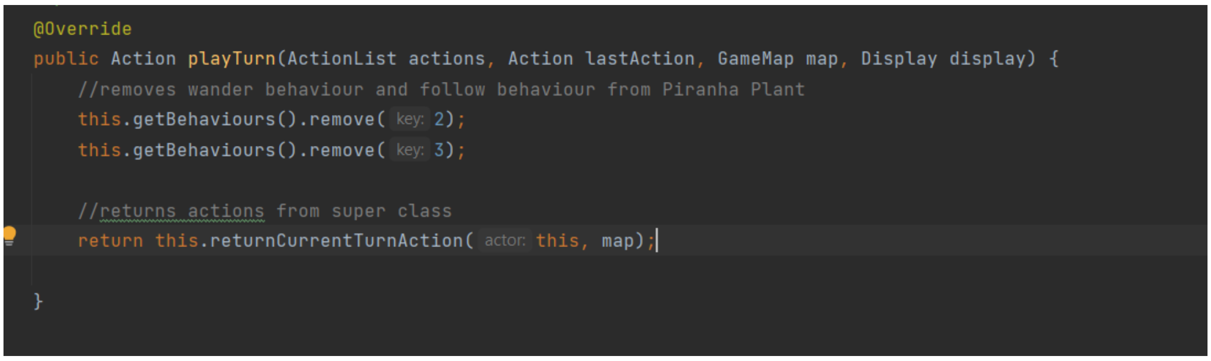
```
public class PiranhaPlant extends EnemyType{
    /**
     * Constructor for piranha plant
     */
    public PiranhaPlant(){
        super( name: "Piranha Plant",  displayChar: 'Y',  hitPoints: 150);
        this.addCapability(Status.PIRANHA_PLANT);
    }
```

It has a display char of 'Y' and hitpoints 150.



```
 */
@Override
protected IntrinsicWeapon getIntrinsicWeapon() { return new IntrinsicWeapon( damage: 90,  verb: "chomps"); }
```

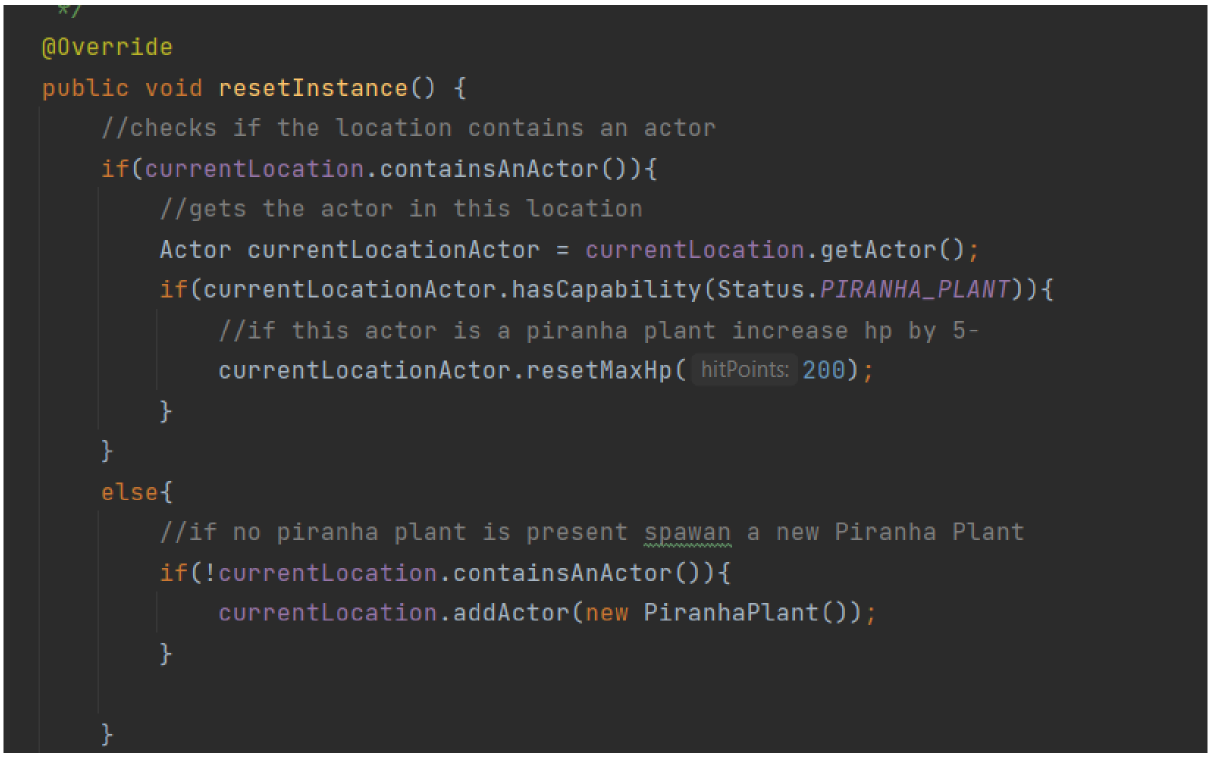
The class also has an intrinsic weapon which has do damage and is named the verb "chomps"



```
@Override
public Action playTurn(ActionList actions, Action lastAction, GameMap map, Display display) {
    //removes wander behaviour and follow behaviour from Piranha Plant
    this.getBehaviours().remove( key: 2);
    this.getBehaviours().remove( key: 3);

    //returns actions from super class
    return this.returnCurrentTurnAction( actor: this, map);

}
```

Since this doesn't follow or attack on in the playTurn Method We get the behaviors from the super class and then removes the follow behavior and the wander behavior

For the reset behavior this is done mostly in the Warp Action class that implements resettable

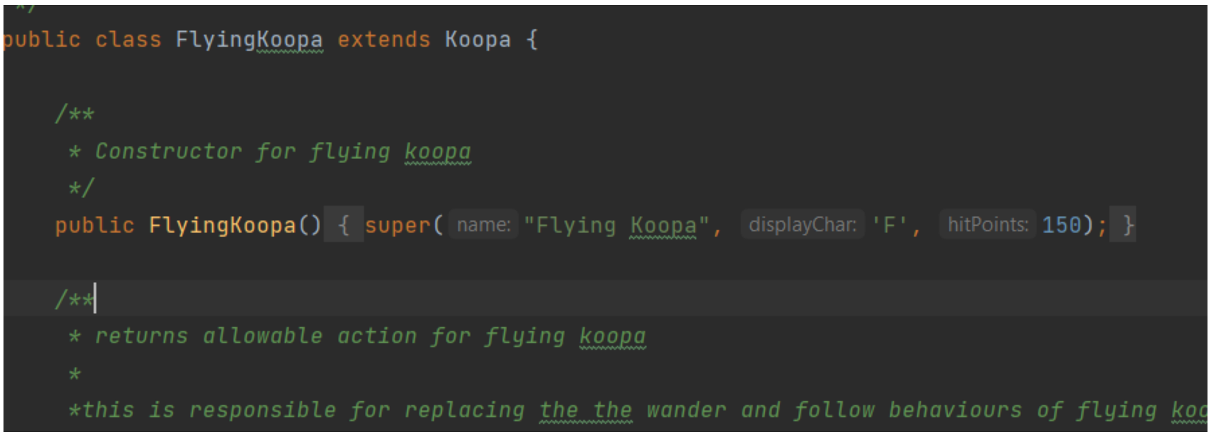


```java
    */
@Override
public void resetInstance() {
    //checks if the location contains an actor
    if(currentLocation.containsAnActor()){
        //gets the actor in this location
        Actor currentLocationActor = currentLocation.getActor();
        if(currentLocationActor.hasCapability(Status.PIRANHA_PLANT)){
            //if this actor is a piranha plant increase hp by 5-
            currentLocationActor.resetMaxHp( hitPoints: 200);
        }
    }
    else{
        //if no piranha plant is present spawan a new Piranha Plant
        if(!currentLocation.containsAnActor()){
            currentLocation.addActor(new PiranhaPlant());
        }

    }
```

In the WarpPipe when the reset action is selected it it checks if the wap pipe locations contins an Actor

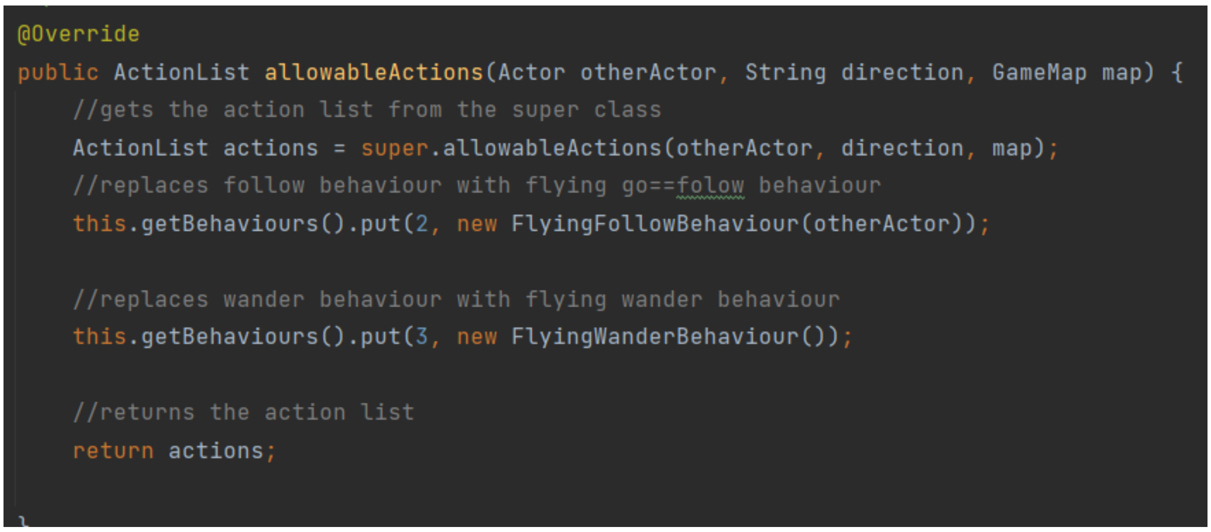
It then checks if this actor is the piranha plant. If it is it resets the piranha plants hp and increases it by 50 points. Else if there is no actor present the Warp Pipe will spawn a new piranha plant in the location

4. Flying Koopa



```java
 */
public class FlyingKoopa extends Koopa {

    /**
     * Constructor for flying koopa
     */
    public FlyingKoopa() { super( name: "Flying Koopa",  displayChar: 'F',  hitPoints: 150); }

    /**
     * returns allowable action for flying koopa
     *
     *this is responsible for replacing the the wander and follow behaviours of flying koo
```

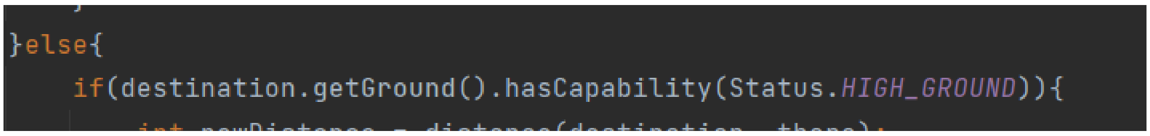
Here The FlyingKoopa extends the Koopa class as both of them share most of the same behavior However I have changed the constructor to have display char ‘F’ and hitpoints 150.



```java
@Override
public ActionList allowableActions(Actor otherActor, String direction, GameMap map) {
    //gets the action list from the super class
    ActionList actions = super.allowableActions(otherActor, direction, map);
    //replaces follow behaviour with flying go==folow behaviour
    this.getBehaviours().put(2, new FlyingFollowBehaviour(otherActor));

    //replaces wander behaviour with flying wander behaviour
    this.getBehaviours().put(3, new FlyingWanderBehaviour());

    //returns the action list
    return actions;

}
```

The difference in Flying Koopa is that it replaces the Wander Behavior and Follow Behavior with Flying Wander and Flying Follow Behavior which enables the flying koopa to traverse across high grounds.

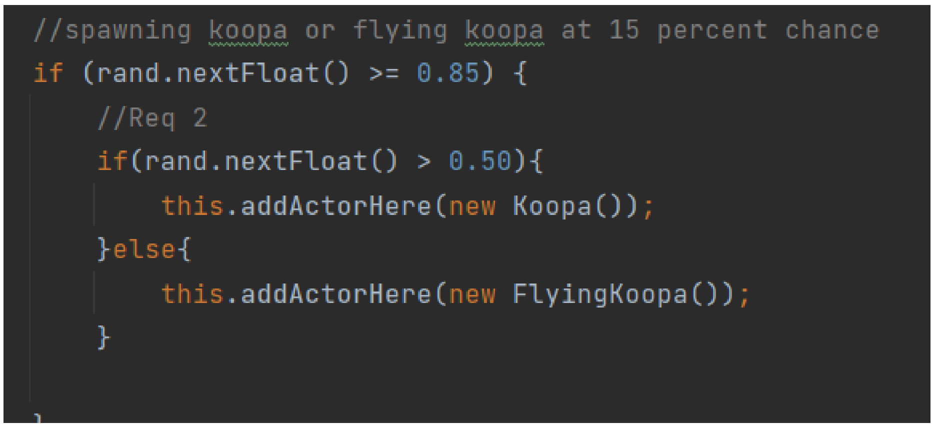
In discussing the Flying Follow Behavior and the Flying Wander Behavior the both of them have a check



```java
}else{
    if(destination.getGround().hasCapability(Status.HIGH_GROUND)){
        int newDistance = distance(destination, there);
```

Shown above which is used to indicate if the location that they cannot enter into is a high ground. If the location is a high ground then an appropriate sequence of events will take place within these behaviors

Spawning Flying Koopa



```java
//spawning koopa or flying koopa at 15 percent chance
if (rand.nextFloat() >= 0.85) {
    //Req 2
    if(rand.nextFloat() > 0.50){
        this.addActorHere(new Koopa());
    }else{
        this.addActorHere(new FlyingKoopa());
    }

}
```

In order to spawn this Flying Koopa the mature tree class has been updated such that once the 15 percent chance has been passed then there is a 50 percent chance to to either spawn the Flying Koopa or Koopa

3.3 Magical Fountain

Requirement 3 states that two fountains would be added into the game, namely a Health Fountain and a Power Fountain. When the player stands atop the fountains, they will be able to retrieve the water from the fountains and store them in a bottle, which is given to them from the start. The player can then decide to consume the water stored in the bottle to enjoy the benefits brought about by their magical properties.

Two classes (*HealthFountain* and *PowerFountain*) were constructed for the fountains, which both extends the abstract *Ground* class as they may make use of the methods defined in that class, this goes along with **Liskov's substitution principle**. These two classes also only govern the fountains, and what the player is allowed to do while they are in the vicinity of the fountains, ensuring that **single responsibility principle** is not breached.

One allowable action for the player while they are standing atop either fountain, is that they will be able to fill up the bottle in their possession with the water from the fountain by executing an instance of the *FillAction* class, which extends the abstract *Action* class. This implements the **dependency inversion principle** by acting as the "middleman" between the player and the items they are about to pick up.

Which brings us to the next part of this deliverables: the *HealthWater* and *PowerWater* classes, which extends the abstract *Item* class. These two classes not only implement the *Consumable* interface, but also the *Bottleable* interface, and care has been taken to ensure that the **interface segregation principle** has not been breached while doing so. The *Bottleable* interface allows the instances of these two classes to be stored in a bottle.

The *Bottle* class takes reference from the *ResetManager* class and is made to be a singleton instance to further enforce abstraction throughout the game in the form of a stack. This means that the water stored in it will follow a "first in last out" (FILO) order, which allows the player to only consume the water that was lasted added to the bottle. When a fill action is executed, either a *HealthWater* or *PowerWater* instance will be pushed into the bottle stack.

Once the player has filled up their bottle with water, they will be able to consume the water through the consume() method inherited from the *Consumable* interface. While the *ConsumeAction* class had been discussed in Assignment 2, some changes have been made since then. Notably, it is no longer responsible for implementing the effects of the consumables, rather it calls the consume() method of relative classes to do that instead. This adheres to the **single responsibility principle** better than the previous implementation as it no longer needs to perform so many instructions.

3.4 Additional debuffs

Requirement 4 states that two new effects should be introduced into the game that provide debuffs to the player in order to keep the balance of ever-increasing maximum hit points after consuming a Super Mushroom. The two effects are the *burned* effect, and the *broken leg* effect. The addition of these two effects required little effort and changes made to the code, which speaks words on how the implementation of the game follows the **open-closed principle** to a certain degree, which allowed for easy extensions to be added.

With the implementation of the *burned* effect, the solution is simply adding the instruction to decrease, or rather increase the maximum hit points of the player by a negative amount for each turn the player stands atop a *Lava* ground type. This adheres to the **open-closed principle** as no modification has been made to the base code of the *Player* class.

Meanwhile, the implementation of the *broken leg* effect was a little more involved, by using the *Status* enumeration to keep track of the failure counts. When the player is plagued by this effect, the choice of selecting and executing a *JumpAction* is impeded by removing these from the allowable action list of the player each turn. The effect lasts for five turns, and all the while will have a counter to keep track of the remaining turns.