

# Problem 9 - Maze-1

---

*Object Oriented Programming*  
*Group CEN 2.2A*  
*II<sup>nd</sup> year, 1<sup>st</sup> semester, Computers*  
Frăsineanu Claudia Andreea







## I. Introduction

The game is kept fairly simple and basic. The player starts the maze at a location and must find his way towards the exit. On his way through the maze he encounters different entities, such as coins, guns, teleporters and monsters. The first three are harmless and can be picked up or used by stepping on them. The monsters are there to catch you and prevent you from exiting the maze. Although powerful, they are not undefeatable. A player can either try to avoid them or fight them head on. In order to win the fight, the player needs a weapon, and multiple can be found throughout the maze.

Each fight has a cost. After defeating an enemy, you will lose the weapon and 100 points from your health. In addition, after a monster is killed, another one will take its place, spawning at its starting position.

One must keep in mind that a truly important rule of the game is that if the player is near an enemy, the player must choose to fight him if it has a weapon. If the player does not have a weapon, the game will be lost automatically.










The commands with which one can operate are:

-  n, s, e, w → each of these represent a direction, and you must choose one in order to go upwards, downwards, to the left or to the right;
-  fight → whenever an enemy is at an adjacent position, the player must fight him. The costs of the fight are explained above;
-  help → prints all the possible commands
-  save → saves the current state of the game
-  load → load a previously saved game
-  quit → stops the game regardless of its state

## II. Modules

### 1. Labyrinth

This class stores the labyrinth through an attribute with the same name. The maze can be seen as a matrix, where each element is one of the following:

-  X → represent a wall; neither players nor enemies can walk through them
-  " " → free space, where players and enemies can move
-  ( → represent the Boomstick, one of the weapons
-  ! → represent a Laser weapon
-  : → represent the Pheonix Blaster
-  \* → represents the coins
-  O → represent the teleporters
-  P → represent the player
-  E → represent the enemies

Inside this class there are 5 methods:

#### a) **loadStartFile(self)**

This function opens the text file, "labyrinth.txt", containing the labyrinth and reads each line. After that the lines are read character with character in order to create a matrix, which is saved in the only variable of the class, labyrinth.

#### b) **loadSavedFile(self, Directions, Player, enemyList, Labyrinth)**

When this function is called, the text file "save.txt" is opened, and read line by line. Each of these lines contains valuable information for the game. Before assigning any of these values they must be converted from string to integer

#### c) **uploadFile(self, Directions, Player, enemyList, Labyrinth)**

It uses the same principle as the previous function, but this time the variables have to be converted into strings that fit the chosen layout.

#### d) **printLabyrinth(self)**

Auxiliary function used to print the labyrinth in a nicer way, in order to give the game an authentic experience.







#### e) **printIntroduction(self)**

This function is called at the beginning and it acts as a "menu" for the game. It gives the player three choices: to start a new game, to load a previously saved game or to quit.

save.txt
Player coordinates
Boomstick
PheonixBlaster
Laser
Coins
Health
Enemy1 coordinates
Enemy2 coordinates
Enemy3 coordinates
Enemy4 coordinates
First row of the maze
-----
Last row of the maze

## 2. Items

This class contains multiple attributes. The interaction between the player and this objects will be discussed in the checkItems() function.

-  coins
-  boomstick
-  pheonixBlaster
-  laser
-  teleporter1
-  teleporter2



The methods found inside the class are:

**a) setItems(self, items)**

This is a setter used when we upload a saved file. Here, items represent an array containing details about each attribute, in this order: boomstick, pheonixBlaster, laser, coins.

**b) allWeapons(self)**

It creates a list containing all the weapons that have been picked up by the player thus far. The attributes containing the weapons have two major states:

-  0, when that weapon has not been picked up
-  any value greater than 0. We can pick up multiple weapons of the same type, so the number will represent the total number of weapons of a specific type we have

**c) hasWeapons(self)**

Checks the value of each weapon. If we the player has it (its value is different than 0), then it will be added to a list. After checking all weapons, the list created will be returned.

**d) resetWeapons(self)**

After fighting an enemy, we will lose one weapon. In case we have multiple weapons, one will be chosen in this particular order: boomstick, pheonixBlaster, laser. Loosing a weapon means we will decrease the value of the attribute by one.

**e) checkItems(self, coordinates, labyrinth)**

This method is called after we move to a new position, which is given through the list containing the two coordinates. The table below illustrates the different symbols and their interaction with the player:





Symbols	Meaning	Interaction
*	coin	add 100 to self.coins
(	boomstick	add 1 to self.boomstick
!	laser	add 1 to self. laser
:	pheonixBlaster	add 1 to self. pheonixBlaster
O	teleporter	moves player to the other teleporter

**f) teleport(self, coordinates, labyrinth)**

Whenever the player decides to step onto one of the teleporters, his position will be moved next to the other one (not on top of it as it would create an infinite loop).

### 3. Directions

The class has three attributes:

-  location → a list containing the location of the player,
-  location → a list containing the location of the player,
-  winLocation → a list containing the location of the exit
-  Itm → an initialization of an object from the class Items. This choice was made to ease the verification of items on the ground.

The methods are:

**a) `uploadSave(self, prevLoc, savedItm)`**

It is used when we want to upload a saved file. the first parameter is a list, containing the coordinates of the last location of the player; savedItm is another list, which is used when calling the function `setItems(savedItm)`, detailed in the Items class.

**b) `moveDown(self, labyrinth)`**

This function determines whether the player can move down or not. If the next location is a wall, a message is printed and nothing else happens. Otherwise we will “erase” our current cell by replace the P with a blank space, we will increase the x coordinate, and check if there are any items that can be picked up (or teleporters). If so, the corresponding interaction is decided by the function `checkItems()` from Items. After that, all that it remains is to replace the cell we wanted to move into with a P, to update the maze.

**c) `moveUp(self, labyrinth)`, `moveRight(self, labyrinth)`, `moveLeft(self, labyrinth)`**



This methods work using the same principle as that of the function `moveDown()` presented above, but instead of increasing the x coordinate, the appropriate coordinate will be updated by increasing or decreasing.

**d) `move(self, new_direction, labyrinth)`**

This method decided which of the above four functions will be called. First we have to check if the value of the parameter *new\_direction* is one of the four allowed (n, s, e, w). If it is not, we will print a message and return. If it is, though, the correct function will be called, depending on the desired direction of movement.

## 4. Player

The attributes of the class are:

-  `health`, which is set by default to 201, in order to allow 2 possible fights with the enemies
-  `gameRunning`, which is set to 1. The moment that the variable will become 0 the game is considered over.




The methods found within this class are:

**a) `setHealth(self, health)`**

This is called when we upload a saved file, and it helps set the correct health of the player

**b) `gameLost(self, playerCoord, enemyCoord, weapon, labyrinth)`**

This function sets the value of the attribute *gameRunning* to 0 if one of the following conditions is met:

-  if the enemy has the same position as the player
-  if the player's health drops below 0
-  if we have no weapon and there is an enemy at an adjacent position

### c) `fight(self, Enemy, Directions, labyrinth)`

As mentioned at the beginning, if a player encounters a monster and has a weapon, he must choose to fight it. The rules of a fight are:

1. the enemy must be at an adjacent position
2. the player must have at least one weapon

After each fight, the following will happen:

- 🧩 the enemy is killed, but another one will take its place, at his spawning position
- 🧩 the weapon becomes used, and disappears from the inventory
- 🧩 the player loses 100 health

If any of the two conditions is not met, the game is automatically lost.

### d) `printStatus(self, Directions)`

This is an auxiliary function, used to print above the maze a few stats, like the location, the score and the weapons found in the player's possession

## 5. Enemy

The attributes of the class are:

- 🧩 `enemyPos`, representing the current position of an enemy,
- 🧩 `startPos`, which represents the position at which a new enemy will respawn after the player kills one.

The methods of this class are:

### a) `resetPosition(self, labyrinth)`

It will clear one enemy from the maze, at the position where we fought it, and will respawn another one at its starting position

### b) `enemyMove(self, labyrinth)`

The movement of the enemy is decided randomly. A value, between 1 and 4 will be generated, and according to it, the enemy will either move up, down, right or left. For each of this 4 possible movements additional methods have been created, similar to the movement of the player.

### c) `enemyMoveUp(self, labyrinth)`

The enemy has two possibilities: if by moving up it encounters a wall or teleporter, it will call the function `enemyMove()` to determine a new movement. This is done in order to give a more random appearance to the monster's movement, as there are a lot of cases where it can run into a wall multiple times. Also, I would like to point out that even this way there is a chance for it to run again into a wall, but, if the function where to call another function of movement, like `enemyMoveDown()` for example, there would be cases where it would be stuck in an infinite loop at some point in the maze and it will result in an error.

Apart from those two types of blocks, the enemies can move anyway in the labyrinth, and if they will step over coins or guns, they will destroy those items.

**d) `enemyMoveUp(self, labyrinth), enemyMoveLeft(self, labyrinth), enemyMoveRight(self, labyrinth)`**

They use the same principle as the function described above, the only difference being in the direction of the movement.

**e) `printEnemyStatus(self)`**

Auxiliary method used to print the location of an enemy. Because there are multiple enemies in the maze, this function is called after a game is over, so as not to overwrite and leave too few space for the actual maze in the terminal.

## 6. main

Within this module all the classes and functions are called in order to successfully run the game. At first, we have to initialize the variables for the Labyrinth, Player and Directions. The Items are initialized in the class Directions, so there is no need to create an object for it in main.

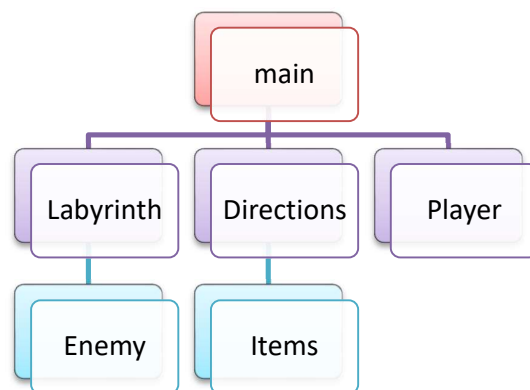
There will be four enemies, and they will be stored in a list, called *enemyList*. Initially it is empty, and it will be filled depending on the user's choice: if it will start a new game, the position of the enemies will be the same as the positions of their spawn, and if it will load a saved game, their positions are determined by their previous locations. It is important to add that for each enemy its starting location is predetermined.

After the variables are loaded and initialized accordingly, a *while* loop will run until the game is considered won or lost. The rules for each of these have been discussed previously.

At each iteration we will check to see if any of the conditions for a lost game have been met. If not, the user can type his command and the program will run the functions in accordance with the command.

## III. Project Hierarchy

The class Directions has one of its attributes an object, of type Items. This decision has been made in order to facilitate the interaction between the player and the objects on the ground. Every time the player is moved to a new position, the function `checkItems()`, from Items, is called through the attribute found in the class Directions.



The class Labyrinth needs the class enemy, because when we want to start a saved game, we have append to the enemy list four objects from the class. Since these objects are not yet created, they need to be instantiated, and thus, we need to use the class Enemy within the class Labyrinth.