2021

World Navigator

TARIQ SHA’BAN

TABLE OF CONTENTS

Abstract ____ 3

Section 1: Code Structure ____ 4

Section 2: Code Specifications ____ 10

Section 2.1: Web Specifications ____ 10

Section 2.1.1: Main Menu ____ 10

Section 2.1.2: Host ____ 13

Section 2.1.3: Lobby Host ____ 16

Section 2.1.4: Join ____ 19

Section 2.1.5: Lobby ____ 22

Section 2.1.6: Game Field ____ 25

Section 2.2: Java Specifications ____ 28

Section 2.2.1: Assets Package ____ 28

Section 2.2.1.1: Maps ____ 28

Section 2.2.1.2: dialogs.json ____ 29

Section 2.2.1.3: Log.log ____ 30

Section 2.2.1.4: StringValues.java ____ 31

Section 2.2.2: Containers Package ____ 32

Section 2.2.2.1 Interfaces Package ____ 32

Section 2.2.2.2 Lootables Package ____ 34

Section 2.2.2.3 Non-Lootables Package ____ 36

Section 2.2.2.4 Container List Class ____ 37

Section 2.2.2.4 Room Class ____ 38

Section 2.2.3: Exceptions Package ____ 39

Section 2.2.3.1 Exception Info Holder ____ 39

Section 2.2.3.2 Illogical Mapping Exception Class ____ 40

Section 2.2.4: Loot Package ____ 41

Section 2.2.4.1 Contents Class ____ 42

Section 2.2.4.2 Chest Key Class ____ 44

Section 2.2.4.3 Door Key Class ____ 45

Section 2.2.4.4 Flashlight Class ____ 46

Section 2.2.4.5 Gold Class _____ *47*
Section 2.2.5: Controls Class _____ *48*
Section 2.2.6: Countdown Class _____ *49*
Section 2.2.7: Games List Class _____ *50*
Section 2.2.8: IO Handler Class _____ *51*
Section 2.2.9: Maze Game Class _____ *52*
Section 2.2.10: Player Class _____ *53*
Section 2.2.11: Store Class _____ *55*
Section 3: Design Patterns Compliance _____ *56*
Section 4: Extension Scenario _____ *58*
Section 5: Further Notes _____ *62*

TABLE OF FIGURES

Figure 1: Object Tree of the Program 4
Figure 2: Visualized Default Map 5
Figure 3: Game Package 6
Figure 4: Webapp Package 7
Figure 5: Webapp Directory 7
Figure 6: Main Menu Page 10
Figure 7: Main Menu JSP 11
Figure 8: Main Manu Servlet 12
Figure 9: Host Page 13
Figure 10: Host Page JSP 14
Figure 11: Host Page Servlet 15
Figure 12: Lobby Host Page 16
Figure 13: Lobby Host JSP 17
Figure 14: Lobby Host Servlet 18
Figure 15: Join Page 19
Figure 16: Join JSP 20
Figure 17: Join Servlet 21
Figure 18: Lobby Page 22
Figure 19: Lobby JSP 23
Figure 20: Lobby Servlet 24
Figure 21: Game Field Page 25
Figure 22: Game Field JSP 26
Figure 23: Game Field Servlet 27
Figure 24: Deafult.json Map File 28
Figure 25: dialogs.json File 29
Figure 26: Log.log file 30
Figure 27: StringValues Class 31
Figure 28: Translate Invocation 31
Figure 29: Translate Invocation 31

Figure 30: Container.java 32
Figure 31: ContainerWithLogicCheck.java 33
Figure 32: Lootables.json 34
Figure 33: Mirror Lootable Container 35
Figure 34: Wall Non-Lootable Container 36
Figure 35: Container List Class 37
Figure 36: Room Class 38
Figure 37: Exception Info Holder 39
Figure 38: Illogical Mapping Exception Class 40
Figure 39: Contents Class 42
Figure 40: Contents Builder Class Usage 43
Figure 41: Contents Object with Empty Loot 43
Figure 42: Chest Key Class 44
Figure 43: Door Key Class 45
Figure 44: Flashlight Class 46
Figure 45: Gold Class 47
Figure 46: Controls Class 48
Figure 47: Countdown Class 49
Figure 48: Games List Class 50
Figure 49: IO Handler Class 51
Figure 50: Maze Game Class 52
Figure 51: Player Class 54
Figure 52: Store Class 55

Abstract

This document will describe and study the requested assignment that revolves around making a world navigator program which complies with some of the clean coding principles.

We will showcase a screenshot of the project itself, as well as some code snapshots.

Section 1: Code Structure

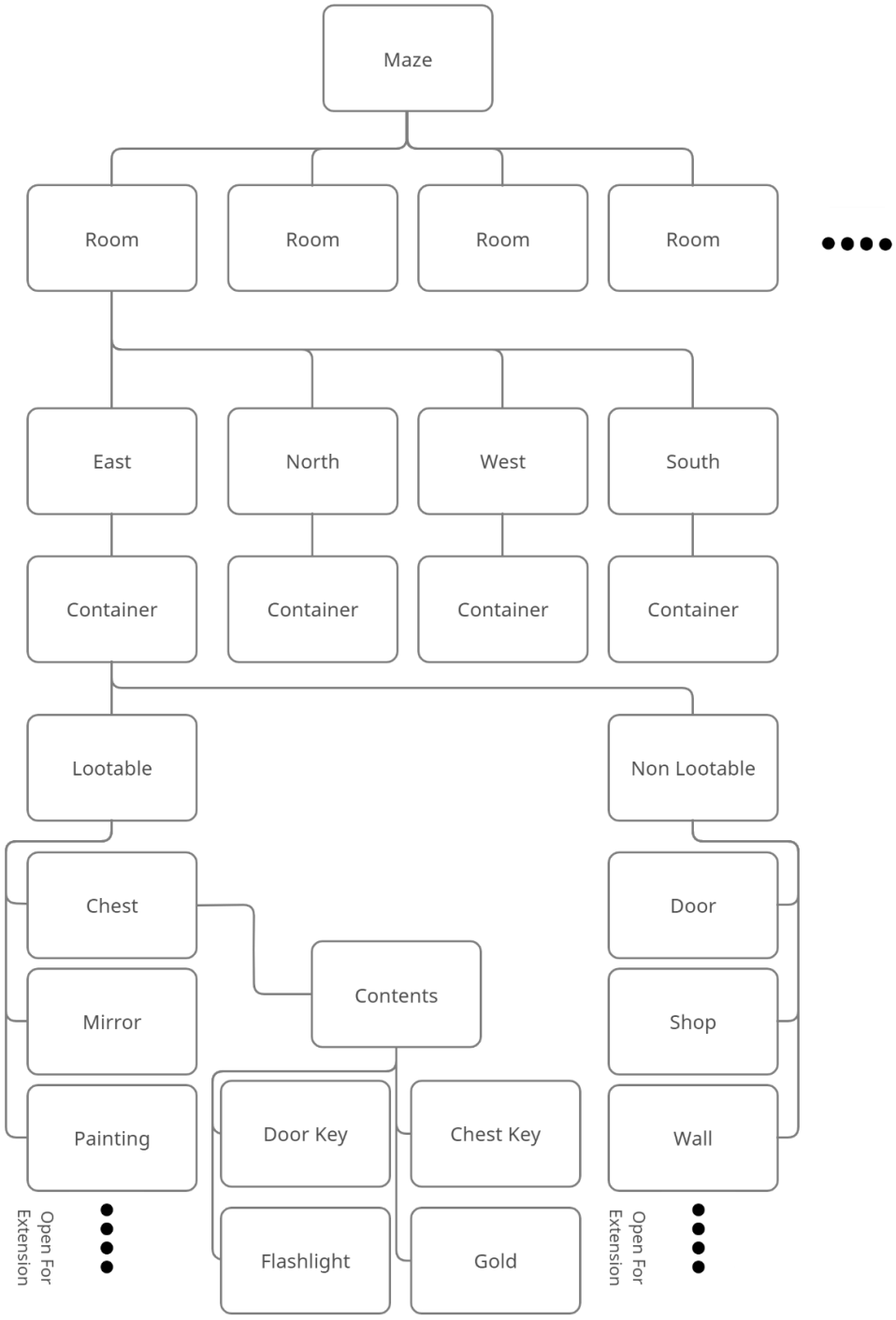


Figure 1: Object Tree of the Program

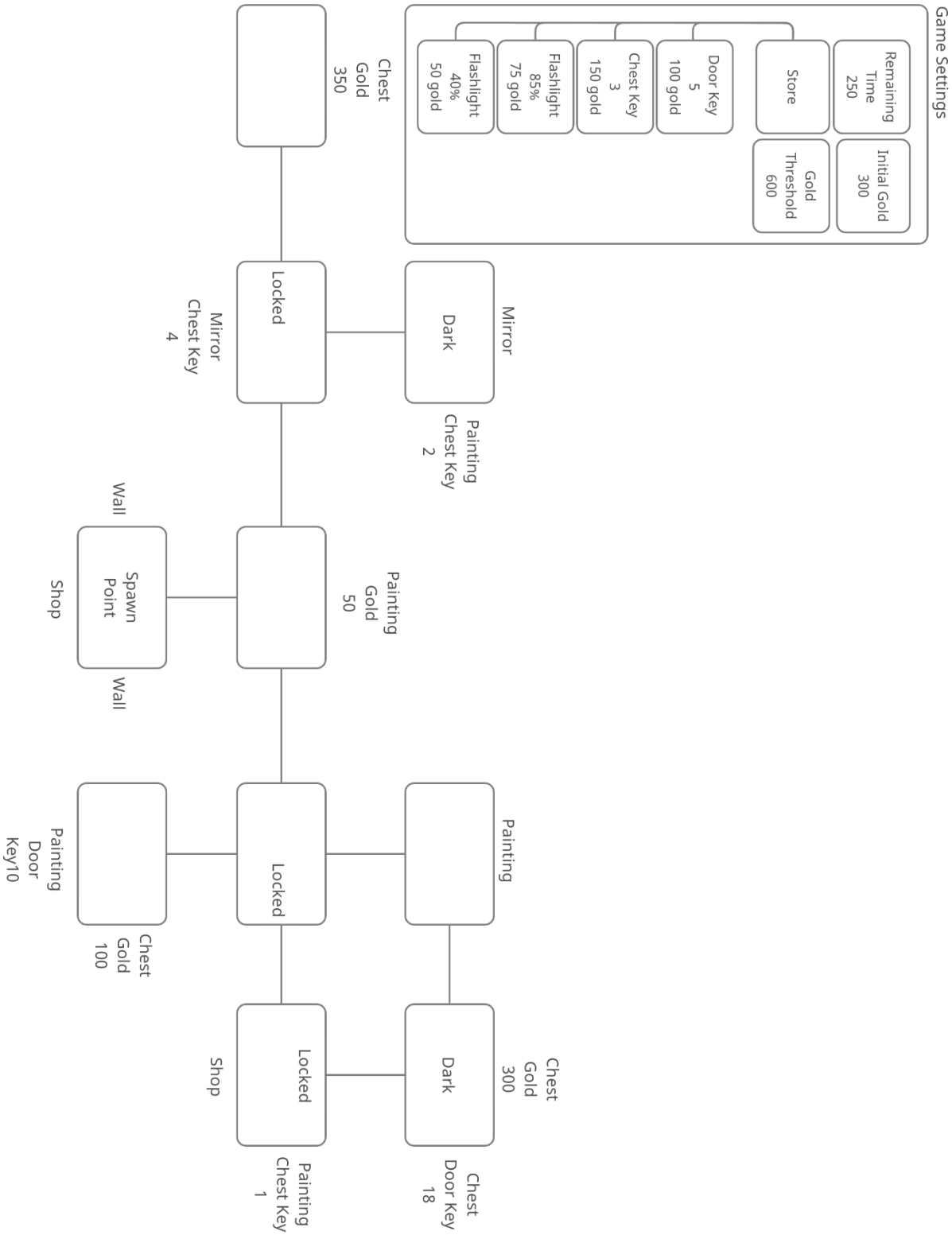


Figure 2: Visualized Default Map

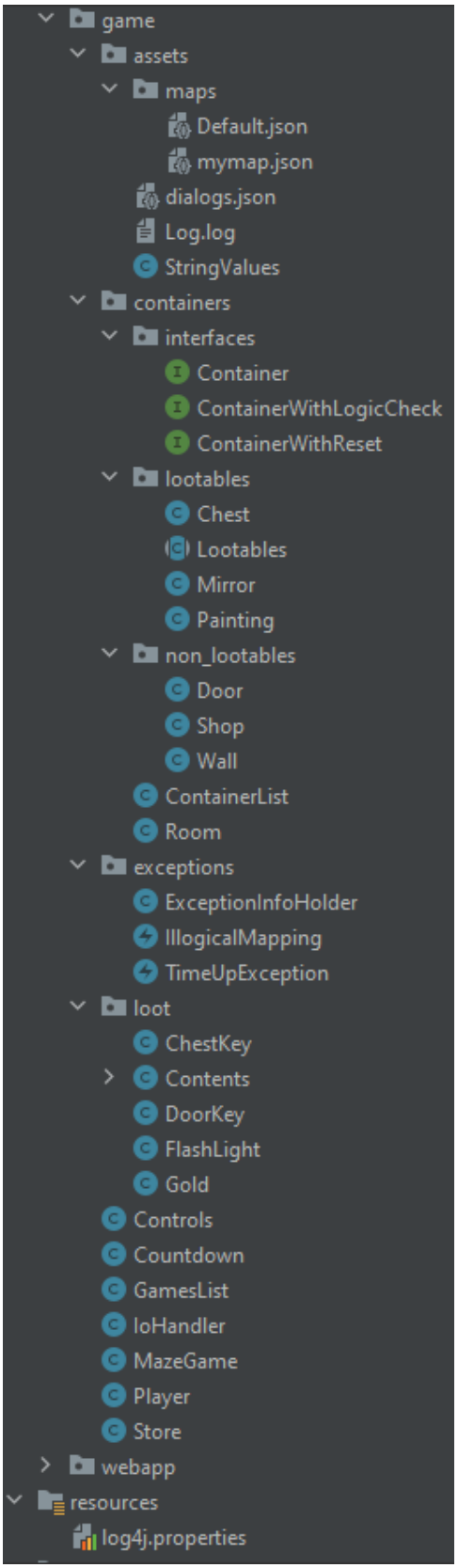


Figure 3: Game Package

Game: A package which contains all the required classes and assets to operate

- assets
  - maps: Store any created map here
  - dialogs.json: Serialized dialogs
  - Logs.log: Logs are stored here
  - StringValues: Reads the dialog json file

- Containers
  - interfaces: Enforce method implementation
  - lootables: Lootable containers
  - non-Lootables: Non-lootable containers
  - ContainerList: binds containers
  - Room: Contains container objects

- exceptions: Holds exception classes

- loot: Houses container's contents

- BatchControls: enforces batch implementation

- Controls: Stores controls binding

- Countdown: Handles the timer as a thread

- GamesList: Stores pending and current games

- IoHandler: handles logs and console streaming

- MazeGame: The main hub of a game's session

- Player: Handles player's choices and inventory

- Store: Contains the store items

Resources: Stores log4j library properties

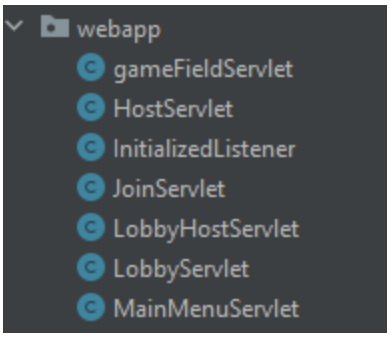


Figure 4: Webapp Package

webapp: A subdirectory in the java folder, handles all callbacks of the JSP files, and communicates with the Java classes and resources

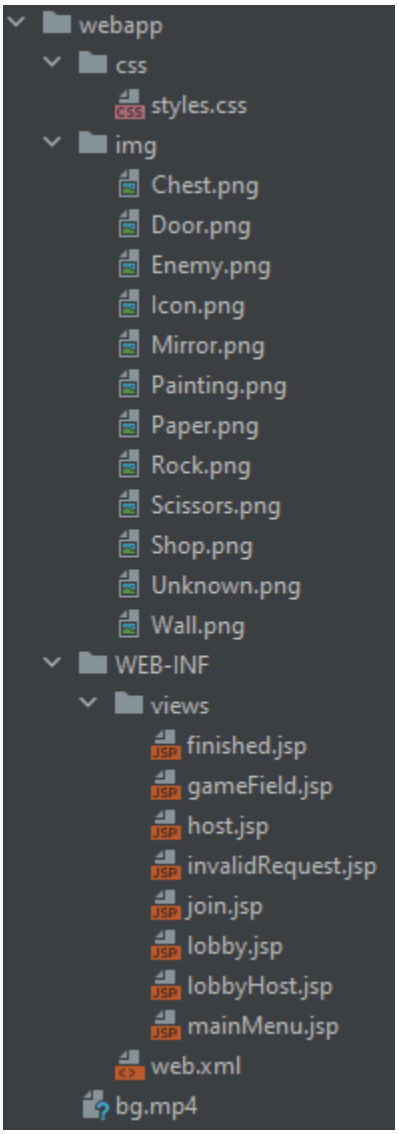


Figure 5: Webapp Directory

webapp: A directory which handles the web view for the user

- css: contains universal styles for all webpages
- img: Image assets
- WEB-INF
  - views: jsp file containing html text
    - finished: Specify victory or defeat
    - gameField: Page where the game starts
    - host: Selects map and create a room
    - invalidRequest: For invalid redirects
    - join: Shows a list of pending games
    - lobby: Shows a list of players in such lobby
    - lobbyHost: Ability to start game
    - mainMenu: Select name and to host or join
  - web.xml: similar to a manifest
  - bg.mp4: a video clip for the background

Each room must have four containers (each denotes a direction).

Any content “loot” which have a value of 0 or less will be ignored.

The standards of creating a map must be followed when making a JSON file inside the map’s directory, any violation will cause an exception to be thrown when deserializing/validating

The guidelines of creating a proper map file are thus:

- The program must have the permission to read the maps directory
- The JSON file must follow the following layout (refer to figure 1 and the default JSON file for clarification):
    - RemainingTime (int)
    - InitialGold (int)
    - GoldThresholdObjective (int)
    - MerchantStore (array of loot with a price entry)
    - Rooms (an array of rooms having containers in each direction and whether the room is dark or not)
    - Each container must have the following entries:
        - Type: denotes if it was a door or a chest for example
        - Which (lootables only): denotes what type of loot is in there (-1 for none)
        - Value (lootables only): denotes the value of the loot (-1 for default)
        - Price (merchant only): denotes the price of the loot
        - To (doors only): denotes which room does this door led to
        - IsLocked (doors only): denotes whether the door is locked or not
- The JSON file must adhere to logical rules, even if the structure is well defined, these logical rules are:
    - The RemainingTime must be at least one minute
    - The InitialGold must not be negative
    - The InitialGold must be less than GoldThresholdObjective
    - The map must have enough gold to ensure winning conditions (startingGold + allContainerGold - StoreItems >= GoldThreshold)
    - The spawn room must not be dark
    - All doors must lead to an existing room
    - All doors must not lead to the same room
    - All doors must have a door in the opposite room
    - All rooms must have at least one door

- All door keys must correspond to an existing door
- All locked doors must have a key
- All chest keys must correspond to an existing chest
- All chests must have a key

Section 2: Code Specifications

This section will further analyze the application in technical perspective, this includes the web aspect (includes the JSP as well as the servlet), and the plain java code.

Section 2.1: Web Specifications

The user will be able to navigate to eight pages, such as thus:

Section 2.1.1: Main Menu

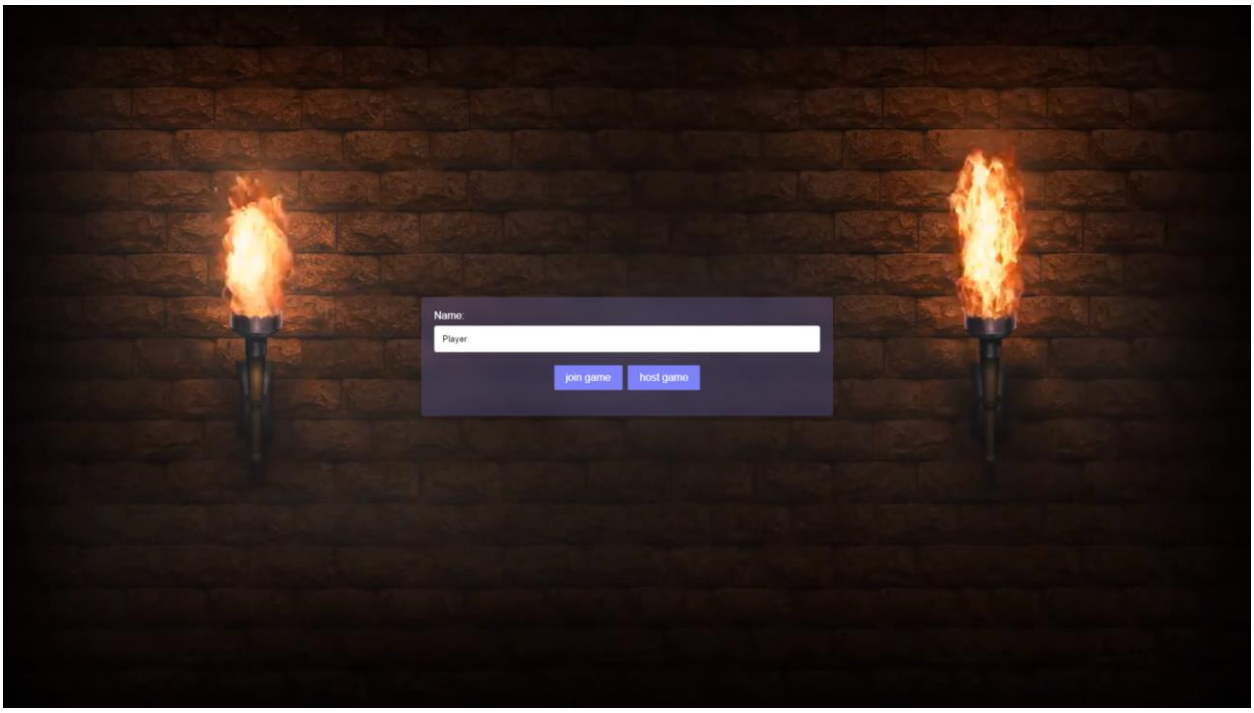


Figure 6: Main Menu Page

The player should enter his name before hosting or joining a game, the name “Player” will be assigned to him if he did not choose a name.

Note that each page includes this background video as well as it inherits from the styles.css file.

```
<html>
   <head>
      <title>Maze Game</title>
      <link rel="icon" href="img/Icon.png">
      <script>
         function addName() {
             document.getElementById("playerJoin").value = document.getElementById("playerField").value;
             document.getElementById("playerHost").value = document.getElementById("playerField").value;
             return true;
         }
      </script>
      <link href="css/styles.css" rel="stylesheet" type="text/css">
   </head>
   <body>
   <video autoplay muted loop id="bg">
     <source src="bg.mp4" type="video/mp4">
   </video>

      <div class="container">
         <label>Name:</label>
         <input type="text" id="playerField" value="Player">
         <div class="formContainer">
            <form class="sideBySideForm" action="/Join" method="GET" onsubmit="return addName()">
               <input type="hidden" id="playerJoin" name="playerName">
               <input class="button" id="join" type="submit" value="join game" />
            </form>
            <form class="sideBySideForm" action="/Host" method="GET" onsubmit="return addName()">
               <input type="hidden" id="playerHost" name="playerName">
               <input class="button" id="hidden" type="submit" value="host game" />
            </form>
         </div>
      </div>
   </body>
</html>
```

Figure 7: Main Menu JSP

Both the buttons (represented by a form each) shares the same text input layout field value.

```java
package webapp;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet(urlPatterns = "/MainMenu")
public class MainMenuServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws IOException, ServletException {

        request.getRequestDispatcher( s: "/WEB-INF/views/mainMenu.jsp").forward(
                request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response) throws IOException, ServletException {

        request.getRequestDispatcher( s: "/WEB-INF/views/mainMenu.jsp").forward(
                request, response);
    }
}
```

Figure 8: Main Manu Servlet

Since this is the first page the user navigates into, there will be no further operations required but to load the page.

Section 2.1.2: Host

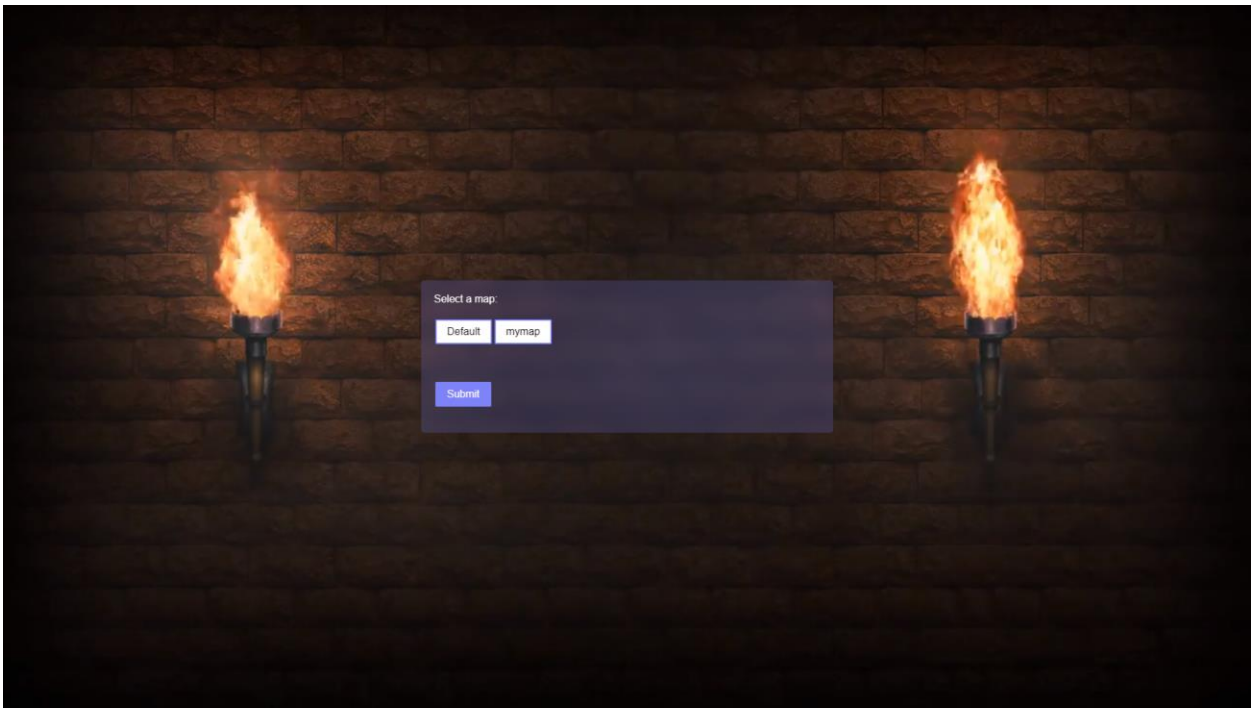


Figure 9: Host Page

This page is intended for users who wish to host their own game, they will be given a choice to select which map (from the maps directory).

If the chosen map happens to be corrupt beyond avoidance, we will apply the fail-fast methodology as show an error page before initializing a new game room, such checks include:

- If the map is deleted immediately after loading this page (highly unlikely).
- If the JSON format is malformed (should not happen unless the file is irresponsibly modified).

```
<html>
   <head>
      <title>Maze Game</title>
      <link rel="icon" href="img/Icon.png">
      <style>
         .selectedMap{visibility: hidden;}
      </style>
      <script>
         function showSelected(x) {
             var hid = document.getElementsByClassName("selectedMap");
             if(hid[0].offsetWidth > 0 && hid[0].offsetHeight > 0) {
                 hid[0].innerHTML="Selected map: "+x;
                 hid[0].style.visibility = "visible";
             }
             document.getElementById("mapChoice").value=x;
         }
         function checkMapChoice(x) {
             var hid = document.getElementsByClassName("selectedMap");
             if(hid[0].innerHTML=="placeholder") {
             alert("Please select a map");
             return false;
             }
             return true;
         }
      </script>
      <link href="css/styles.css" rel="stylesheet" type="text/css">
   </head>
   <body>
   <video autoplay muted loop id="bg">
     <source src="bg.mp4" type="video/mp4">
   </video>
      <div class="container">
         <label>Select a map:</label><br><br>
         ${maps}
         <p class="selectedMap">placeholder</p>
         <form action="/LobbyHost" method="GET" onsubmit="return checkMapChoice()">
            <input type="hidden" name="playerName" value="${playerName}">
            <input type="hidden" id="mapChoice" name="mapChoice" value="">
            <input id="launch" type="submit"/>
         </form>
      </div>
      </div>
   </body>
</html>
```

Figure 10: Host Page JSP

Note that data filled from previous pages is being transmitted to the next one

```java
package webapp;

import game.GamesList;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.List;

@WebServlet(urlPatterns = "/Host")
public class HostServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws IOException, ServletException {

        List<String> list = GamesList.getMaps();
        StringBuilder mapsText = new StringBuilder();
        for (String string : list)
            mapsText.append("<button id=").append(string).append(" type=button onclick=showSelected('").append(string).append("')>").append(string).append("</button>");


        request.setAttribute( s: "maps", mapsText);
        request.setAttribute( s: "playerName", request.getParameter( s: "playerName"));
        request.getRequestDispatcher( s: "/WEB-INF/views/host.jsp").forward(
                request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response) throws IOException, ServletException {
        request.getRequestDispatcher( s: "/WEB-INF/views/host.jsp").forward(
                request, response);
    }
}
```

Figure 11: Host Page Servlet

The map button list is generated from the servlet, each button has the value of the map name (excluded the extension since it is already known that it is in JSON format).

Section 2.1.3: Lobby Host

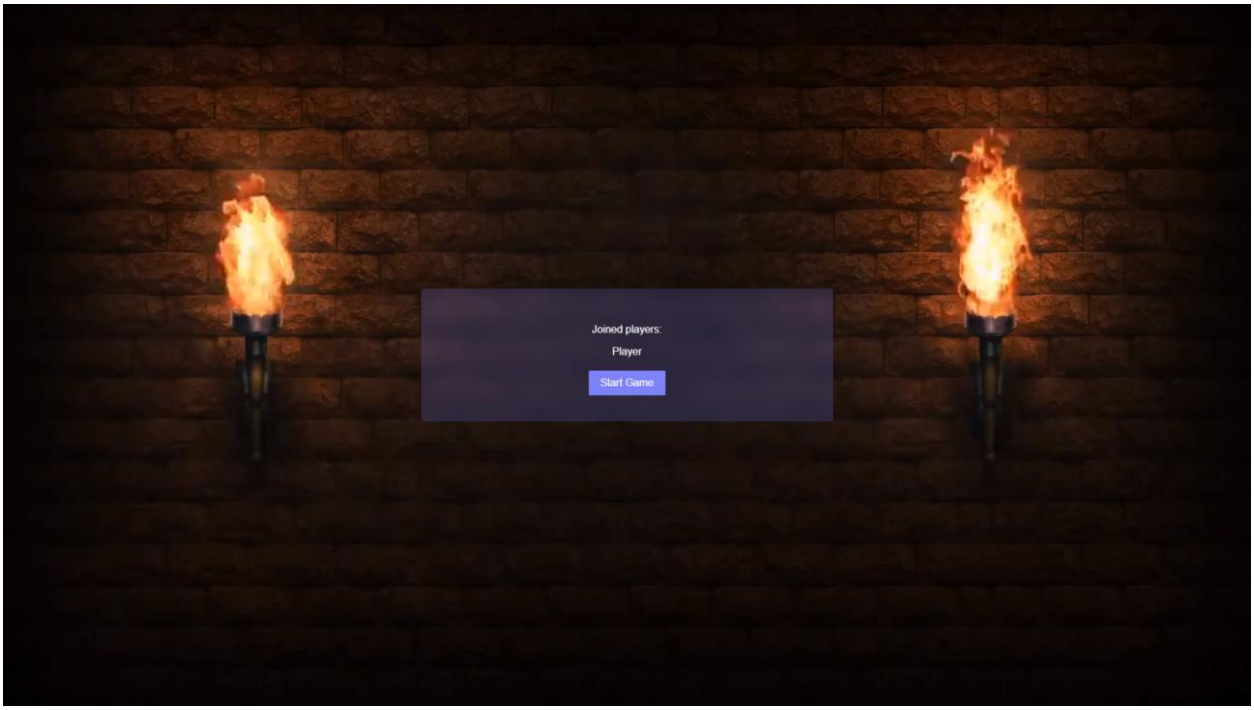


Figure 12: Lobby Host Page

This page is the lobby of the host, the host can view the joined player (by refreshing the page in an interval of five seconds), the host can proceed to start the game at any state, and it will broadcast the game start event to all players. The host can also start the game alone.

```
<html>
   <head>
      <title>Maze Game</title>
      <link rel="icon" href="img/Icon.png">
      <script>
         window.onload = function () {
             setTimeout(function() { document.getElementById("form").submit(); }, 5000);
         };
      </script>
      <link href="css/styles.css" rel="stylesheet" type="text/css">
   </head>
   <body>
   <video autoplay muted loop id="bg">
     <source src="bg.mp4" type="video/mp4">
   </video>
      <div class="container">
         <div class="formContainer">
            <br>
            <p>Joined players: </p>
            ${players}
            <form id="form" action="/LobbyHost" method="GET">
               <input type="hidden" name="gameID" value="${gameID}">
               <input type="hidden" name="mapChoice" value="${mapChoice}">
               <input type="hidden" name="playerID" value="${playerID}">
               <input type="hidden" name="playerName" value="${playerName}">
            </form>
            <form action="/GameField" method="GET">
               <input type="hidden" name="gameID" value="${gameID}">
               <input type="hidden" name="mapChoice" value="${mapChoice}">
               <input type="hidden" name="playerID" value="${playerID}">
               <input type="hidden" name="playerName" value="${playerName}">
               <input id="launch" type="submit" value="Start Game"/>
            </form>
         </div>
      </div>
   </body>
</html>
```

Figure 13: Lobby Host JSP

The page saves the player name and ID, as well as the game map choice and ID

```java
@Override
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) throws IOException, ServletException {
    String gameID;
    String mapChoice = request.getParameter( s: "mapChoice");
    String playerID = "";
    String playerName = request.getParameter( s: "playerName");
    List<String> players = new ArrayList<>();
    boolean filled = request.getParameter( s: "gameID") != null;

    if (request.getParameter( s: "gameID") == null)
        gameID = GamesList.gamesList.get(GamesList.hostGame()).getId();
    else
        gameID = request.getParameter( s: "gameID");


    if (gameID != null)
        for (MazeGame mazeGame : GamesList.gamesList)
            if (mazeGame.getId().equals(gameID) && !mazeGame.isStarted()) {
                try {
                    mazeGame.decodeJSON(mapChoice);
                } catch (IllogicalMapping | FileNotFoundException exception) {
                    exception.printStackTrace();
                }
                if (request.getParameter( s: "playerID") == null) {
                    Player player = new Player(playerName, mazeGame);
                    playerID = player.getId();
                    mazeGame.getPlayers().put(player, false);
                } else playerID = request.getParameter( s: "playerID");
                for (Player playerInstance : mazeGame.getPlayers().keySet())
                    players.add(playerInstance.getName());
            }

    StringBuilder playersText = new StringBuilder();
    for (String string : players)
        playersText.append("<p>").append(string).append("</p>");


    request.setAttribute( s: "gameID", gameID);
    request.setAttribute( s: "mapChoice", mapChoice);
    request.setAttribute( s: "playerID", playerID);
    request.setAttribute( s: "playerName", playerName);
    request.setAttribute( s: "players", playersText);


    if (!filled) {
        response.sendRedirect( s: request.getServletPath()
                + "?gameID=" + gameID + "&"
                + "mapChoice=" + mapChoice + "&"
                + "playerID=" + playerID + "&"
                + "playerName=" + playerName);
    } else request.getRequestDispatcher( s: "/WEB-INF/views/lobbyHost.jsp").forward(
            request, response);
}
```

Figure 14: Lobby Host Servlet

The servlet creates the map as well as the player ID.

Section 2.1.4: Join

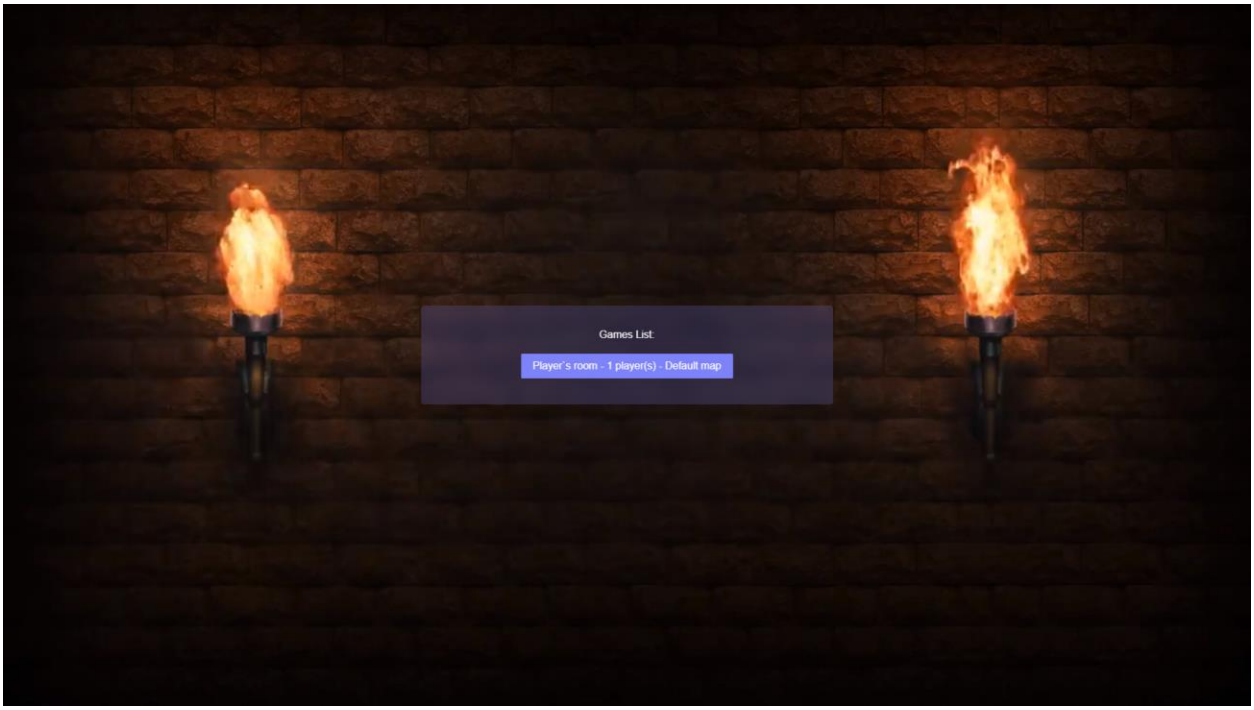


Figure 15: Join Page

User should be able to join the desired available room here (if any exists).

The list is fetched from the games list that have not started yet, it indicated the following information:

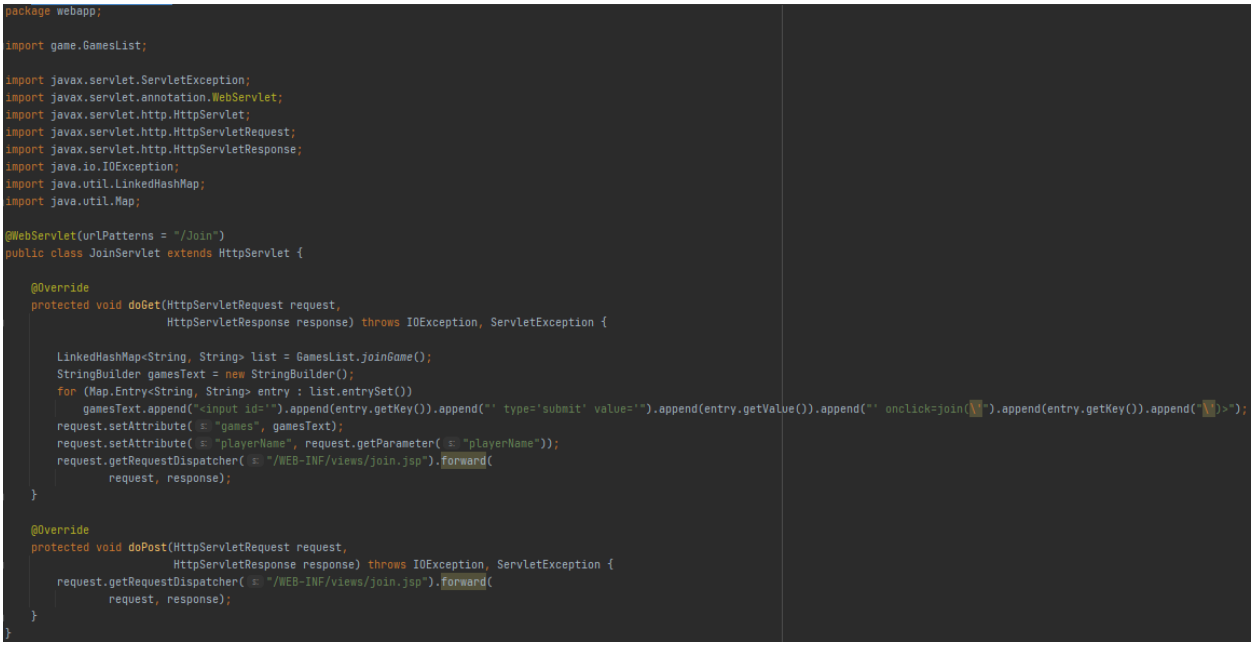
- Host name
- Map type
- Joined players (including the host)
- Maximum capacity (calculated by counting the lit rooms)

```
<html>
   <head>
      <title>Maze Game</title>
      <link rel="icon" href="img/Icon.png">
      <script>
         window.onload = function () {
             setTimeout(function() { document.getElementById("form").submit(); }, 5000);
         };

         function join(x) {
         document.getElementById('gameID').value=x;
         }
      </script>
      <link href="css/styles.css" rel="stylesheet" type="text/css">
   </head>
   <body>
   <video autoplay muted loop id="bg">
     <source src="bg.mp4" type="video/mp4">
   </video>
      <div class="container">
         <div class="formContainer">
            <p>Games List:</p>
            <form action="/Lobby" method="GET">
               <input type="hidden" id="gameID" name="gameID" value="">
               <input type="hidden" name="playerName" value="${playerName}">
               ${games}
            </form>
         </div>
      </div>
      <form id="form" action="/Join" method="GET">
         <input type="hidden" name="playerName" value="${playerName}">
      </form>
   </body>
</html>
```

Figure 16: Join JSP

The page saves the player's name as well shows the room list fetched from the JSP.



```java
package webapp;

import game.GamesList;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.LinkedHashMap;
import java.util.Map;

@WebServlet(urlPatterns = "/Join")
public class JoinServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws IOException, ServletException {

        LinkedHashMap<String, String> list = GamesList.joinGame();
        StringBuilder gamesText = new StringBuilder();
        for (Map.Entry<String, String> entry : list.entrySet())
            gamesText.append("<input id='").append(entry.getKey()).append("' type='submit' value='").append(entry.getValue()).append("' onclick=join(\"").append(entry.getKey()).append("\")>");
        request.setAttribute( s: "games", gamesText);
        request.setAttribute( s: "playerName", request.getParameter( s: "playerName"));
        request.getRequestDispatcher( s: "/WEB-INF/views/join.jsp").forward(
                request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response) throws IOException, ServletException {
        request.getRequestDispatcher( s: "/WEB-INF/views/join.jsp").forward(
                request, response);
    }
}
```

Figure 17: Join Servlet

Retrievers available games from the linked HashMap.

Section 2.1.5: Lobby

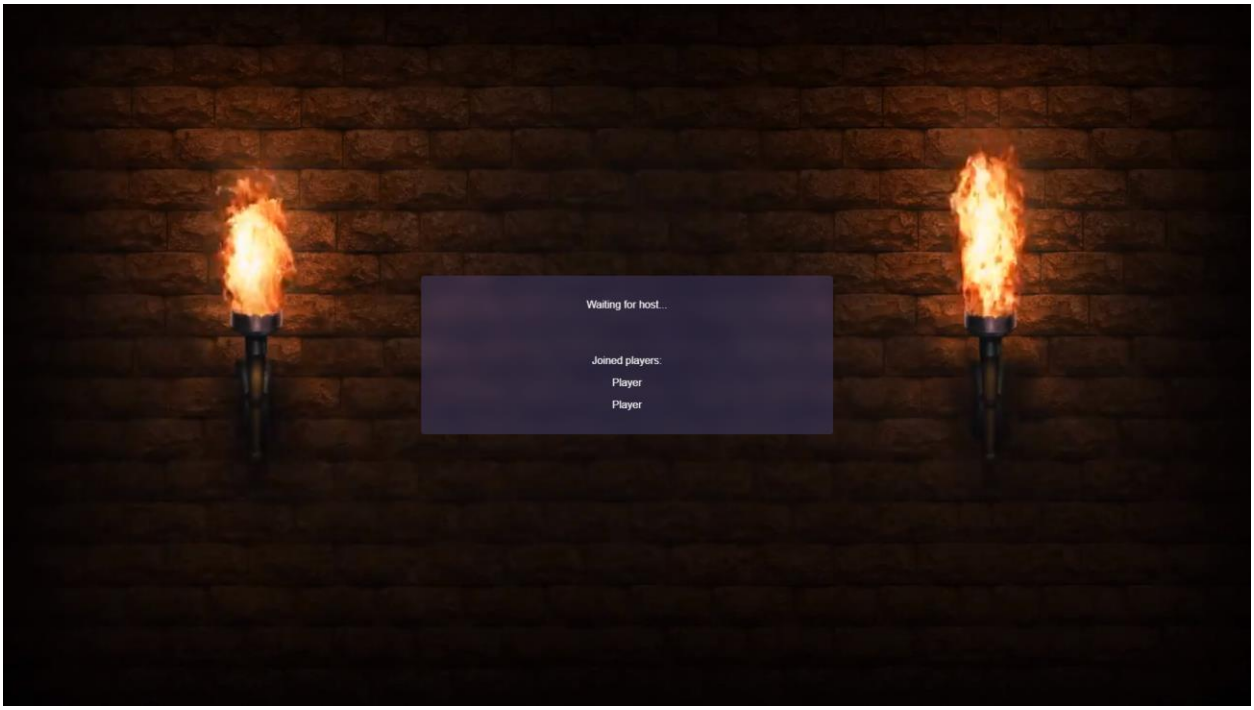


Figure 18: Lobby Page

Users who joined the room can view the joined players (host included), the page will auto refresh every five seconds to check whether the host started the game or not.

```
<html>
   <head>
      <title>Maze Game</title>
      <link rel="icon" href="img/Icon.png">
      <script>
         window.onload = function () {
             setTimeout(function() { document.getElementById("form").submit(); }, 5000);
         };
      </script>
      <link href="css/styles.css" rel="stylesheet" type="text/css">
   </head>
   <body>
   <video autoplay muted loop id="bg">
     <source src="bg.mp4" type="video/mp4">
   </video>
      <div class="container">
         <div class="formContainer">
            <p>Waiting for host...</p>
            <br><br>
            <p>Joined players:</p>
            ${players}
            <form id="form" action="/Lobby" method="GET">
               <input type="hidden" name="gameID" value="${gameID}">
               <input type="hidden" name="mapChoice" value="${mapChoice}">
               <input type="hidden" name="playerID" value="${playerID}">
               <input type="hidden" name="playerName" value="${playerName}">
            </form>
         </div>
      </div>
   </body>
</html>
```

Figure 19: Lobby JSP

The page will automatically refresh every five seconds

```java
@Override
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) throws IOException, ServletException {
    //System.out.println(request.getLocalAddr());
    //System.out.println(request.getRemoteAddr());

    String gameID = request.getParameter( s: "gameID");
    String playerID = "";
    String mapChoice = request.getParameter( s: "mapChoice");
    String playerName = request.getParameter( s: "playerName");
    List<String> players = new ArrayList<>();
    boolean filled = request.getParameter( s: "playerID") != null;

    if (gameID != null)
        for (MazeGame mazeGame : GamesList.gamesList)
            if (mazeGame.getId().equals(gameID) && !mazeGame.isStarted()) {
                if (request.getParameter( s: "playerID") == null) {
                    Player player = new Player(playerName, mazeGame);
                    playerID = player.getId();
                    mazeGame.getPlayers().put(player, false);
                    mapChoice = mazeGame.getMap();
                } else playerID = request.getParameter( s: "playerID");
                for (Player playerInstance : mazeGame.getPlayers().keySet())
                    players.add(playerInstance.getName());
                break;
            } else if (mazeGame.getId().equals(gameID) && mazeGame.isStarted()) {
                request.getRequestDispatcher( s: "GameField").forward(
                        request, response);
                return;
            }

    StringBuilder playersText = new StringBuilder();
    for (String string : players)
        playersText.append("<p>").append(string).append("</p>");

    request.setAttribute( s: "gameID", gameID);
    request.setAttribute( s: "mapChoice", mapChoice);
    request.setAttribute( s: "playerID", playerID);
    request.setAttribute( s: "playerName", playerName);
    request.setAttribute( s: "players", playersText);

    if (!filled) {
        response.sendRedirect( s: request.getServletPath()
                + "?gameID=" + gameID + "&"
                + "mapChoice=" + mapChoice + "&"
                + "playerID=" + playerID + "&"
                + "playerName=" + playerName);
    } else request.getRequestDispatcher( s: "/WEB-INF/views/lobby.jsp").forward(
            request, response);
}
```

Figure 20: Lobby Servlet

Player ID will be instantiated (if it does not exist).

Section 2.1.6: Game Field

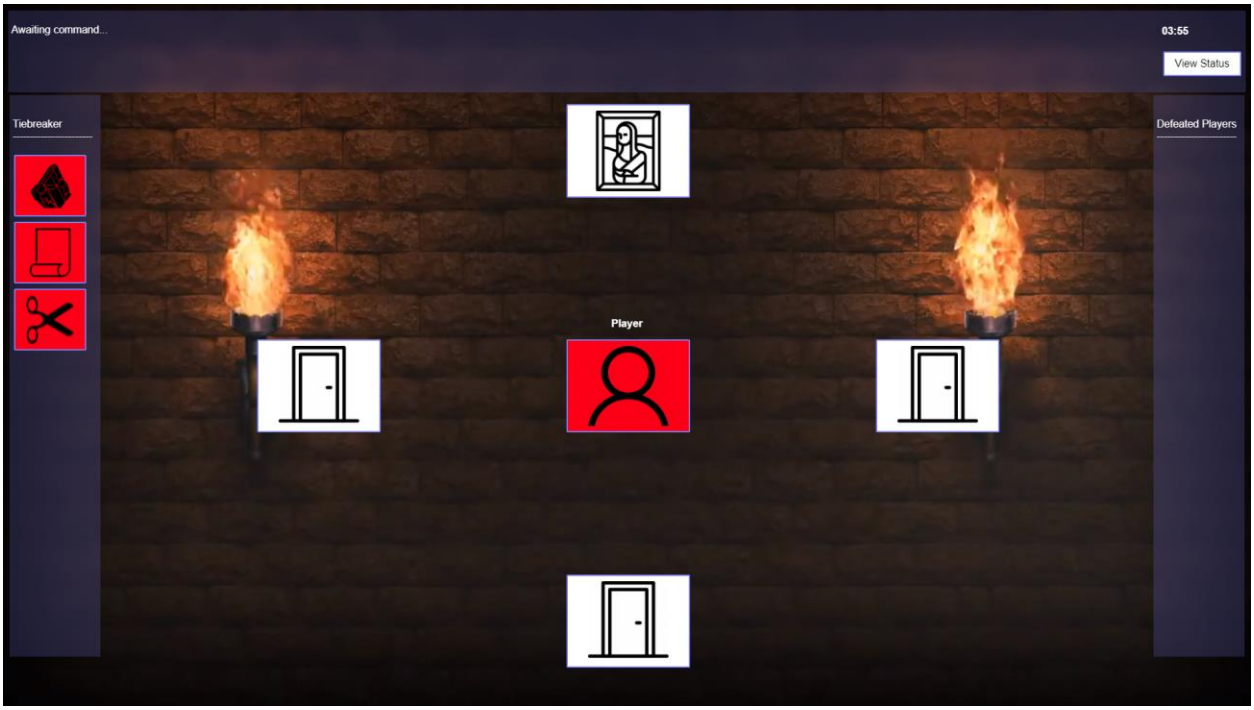


Figure 21: Game Field Page

The main game starts here. Users can move, loot containers, buy from merchants and engage opponents, a tie breaker will start if both players have the same amount of gold.

Users cannot loot or leave the room once an opponent reach their area.

Players can view they action's results from the top bar. Also, they can leave the game as well as view their status using a button.

Players can view defeated players from the right-side bar.

Note that if a player unlocked a door using a key, opponents will be able to access it without having to have a key, this is made to prevent other opponents who did not have the chance to get the key from being trapped.

```
    <script>
      function startTimer(duration, display) {
          var timer = duration, minutes, seconds;
          setInterval(function () {
              minutes = parseInt(timer / 60, 10);
              seconds = parseInt(timer % 60, 10);

              minutes = minutes < 0 ? 0 : minutes;
              seconds = seconds < 0 ? 0 : seconds;

              minutes = minutes < 10 ? "0" + minutes : minutes;
              seconds = seconds < 10 ? "0" + seconds : seconds;

              display.textContent = minutes + ":" + seconds;

              timer--;


          }, 1000);
      }

      window.onload = function () {
          var remaining = ${time}-1;
          display = document.querySelector('#time');
          startTimer(remaining, display);

          setTimeout(function() { document.getElementById("form").submit(); }, 5000);
      };
    </script>
    ${flashlight}
   <link href="css/styles.css" rel="stylesheet" type="text/css">
</head>
<body>
<video autoplay muted loop id="bg">
  <source src="bg.mp4" type="video/mp4">
</video>
    <div class="sticky" style="overflow: auto;">
      <div style="display:inline-block;">
        <p>${message}</p>
      </div>
      <div style="float: right;">
        <br><strong id="time"></strong><br><br>
        <form action="/GameField" method="GET">
          <input type="hidden" name="gameID" value="${gameID}">
          <input type="hidden" name="mapChoice" value="${mapChoice}">
          <input type="hidden" name="playerID" value="${playerID}">
          <input type="hidden" name="playerName" value="${playerName}">
          <button type="submit" name="command" value="t">View Status</button>
        </form>
      </div>
    </div>
    <div class="stickyRight" style="overflow: auto;">
      <div style="display:inline-block;">
        <p>Defeated Players<br>----------------------</p>
      </div>
      ${defeatedPlayers}
    </div>
    <form action="/GameField" method="GET">
      <input type="hidden" name="gameID" value="${gameID}">
      <input type="hidden" name="mapChoice" value="${mapChoice}">
      <input type="hidden" name="playerID" value="${playerID}">
      <input type="hidden" name="playerName" value="${playerName}">
      ${tiebreakerSidebar}
```

Figure 22: Game Field JSP

Shows the main outlines of the page.

```
for (MazeGame mazeGame : GamesList.gamesList)
    if (mazeGame.getId().equals(gameId)) {
        mazeGame.setStarted(true);
        mazeGame.startGame();
        player = mazeGame.getPlayers().keySet().stream().filter(u -> u.getId().equals(playerID)).collect(Collectors.toList()).get(0);
        player.startGame();
        if (command != null) {
            if (Player.isContainerAShop(player, command.toUpperCase().charAt(0)) && choice == null) {
                List<StringBuilder> list = Shop.viewShopItems(mazeGame, player);
                StringBuilder shopItemsText = new StringBuilder();
                for (StringBuilder string : list)
                    shopItemsText.append("<button name=choice value=")
                            .append(string.toString().charAt(0))
                            .append(" type=submit>")
                            .append(string.toString()).append("</button>");
                request.setAttribute( s: "shopItems", shopItemsText);
            }

            player.move(command.charAt(0));

        } else {
            if (choice != null) {
                Shop.makeTransaction(mazeGame, player, Integer.parseInt( s: choice.charAt(0) + ""));
            } else if(player.isOpponentInRoom() && player.getTieBreakerChoice()==-1) {
                player.move(Controls.getEngageKey());
            } else if(player.isOpponentInRoom()) {
                player.move(Controls.getRandomTieBreakerChoice());
            }
        }

        if (player.getInventory().getGold().getAmount() >= mazeGame.getGoldThresholdObjective()) {
            player.clearBuffer();
            player.printStatus(mazeGame);
            request.setAttribute( s: "message", player.getMessageBuffer());
            request.setAttribute( s: "result",  o: "<h1><font color=\"green\">YOU ARE VICTORIOUS!</font></h1>");
            request.getRequestDispatcher( s: "/WEB-INF/views/finished.jsp").forward(
                    request, response);
            return;
        }

        if (mazeGame.getRemainingTime() < 0 || Boolean.TRUE.equals(mazeGame.getPlayers().get(player))) {
            player.clearBuffer();
            player.printStatus(mazeGame);
            request.setAttribute( s: "message", player.getMessageBuffer());
            request.setAttribute( s: "result",  o: "<h1><font color=\"red\">YOU HAVE BEEN DEFEATED!</font></h1>\n");
            request.getRequestDispatcher( s: "/WEB-INF/views/finished.jsp").forward(
                    request, response);
            return;
        }

        if (player.getLocation().isDark())
            flashlight = "<style> :root {\n" +
                    "  cursor: none;\n" +
                    "  --cursorX: 50vw;\n" +
                    "  --cursorY: 50vh;\n" +
                    "  --z-index: 0;\n" +
                    "}\n" +
                    ":root:before {\n" +
                    "  content: '';\n" +
                    "  display: block;\n" +
                    "  width: 100%;\n" +
                    "  height: 100%;\n" +
                    "  z-index: 5;\n" +
```

Figure 23: Game Field Servlet

Handles commands given from the JSP file.

Section 2.2: Java Specifications

Section 2.2.1: Assets Package

A package with handles serializing/deserializing operations, as well as storing them.

Section 2.2.1.1: Maps

Contains all the map's specifications in a JSON format.

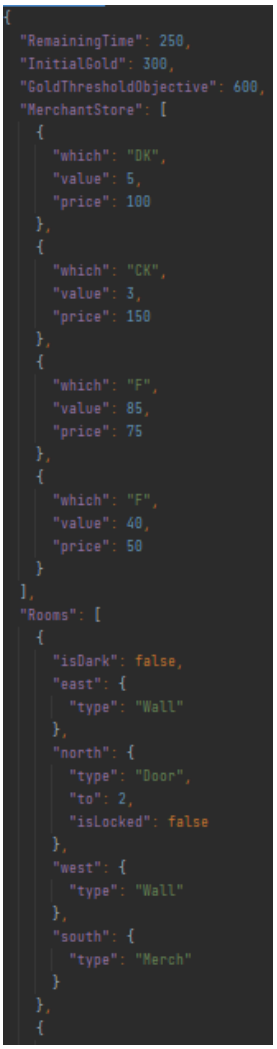


Figure 24: Deafult.json Map File

Section 2.2.1.2: dialogs.json

Contains dialogs that are shown to either the user or logged into the log file.

```
"UI_AWAITING_COMMAND": "Awaiting command...",
"UI_INVALID_COMMAND": "Invalid command",
"UI_INVALID_SELECTION": "Invalid selection",
"UI_FLASHLIGHT_CHARGE_IS": "Current flashlight charge is: ",
"UI_OBJECTIVE_COMPLETE": "I think I have more than enough for today",
"UI_BLIND": "I cant see, I'll need a flashlight to enter that room",
"UI_DOOR_UNLOCKED": "I've unlocked the door using key number ",
"UI_DOOR_LOCKED": "It's locked, I'll need door key number ",
"UI_OPPOSITE_DOOR_UNLOCKED": "I've unlocked the opposite door using key number ",
"UI_FIGHT_HAPPENING": "There is currently a fight happening",
"UI_OPPOSITE_DOOR_LOCKED": "It's locked from the opposite room, I'll need door key number ",
"UI_CHEST_LOCKED": "It's locked, I'll need chest key number ",
"UI_NO_LOOT": "Got nothing here",
"UI_GOLD_FOUND": [
  "Found ",
  ", now I've got ",
  " gold"
],
```

Figure 25: dialogs.json File

Section 2.2.1.3: Log.log

Shows server start-up times, as well as shows exceptions when they arise.

```
2021-02-15 22:07:55 IoHandler:23 - Establishing Server...
2021-02-15 22:50:01 IoHandler:23 - Establishing Server...
2021-02-15 22:55:52 IoHandler:23 - Establishing Server...
2021-02-15 22:58:10 IoHandler:23 - Establishing Server...
2021-02-16 13:25:40 IoHandler:23 - Establishing Server...
2021-02-17 15:29:05 IoHandler:23 - Establishing Server...
2021-02-18 13:52:23 IoHandler:23 - Establishing Server...
2021-02-18 13:58:10 IoHandler:23 - Establishing Server...
2021-02-18 14:04:37 IoHandler:23 - Establishing Server...
2021-02-18 14:07:36 IoHandler:23 - Establishing Server...
2021-02-18 14:08:11 IoHandler:23 - Establishing Server...
2021-02-18 14:08:34 IoHandler:23 - Establishing Server...
2021-02-18 14:16:54 IoHandler:23 - Establishing Server...
2021-02-18 14:25:40 IoHandler:23 - Establishing Server...
2021-02-18 14:26:36 IoHandler:23 - Establishing Server...
2021-02-18 14:28:52 IoHandler:23 - Establishing Server...
2021-02-18 14:29:49 IoHandler:23 - Establishing Server...
2021-02-18 14:35:23 IoHandler:23 - Establishing Server...
2021-02-18 14:45:04 IoHandler:23 - Establishing Server...
2021-02-18 14:45:14 IoHandler:23 - Establishing Server...
2021-02-18 15:59:54 IoHandler:23 - Establishing Server...
2021-02-18 15:59:57 IoHandler:23 - Validate the logic of the JSON file map, remaining time cannot be less than a minute
```

Figure 26: Log.log file

Section 2.2.1.4: StringValues.java

```java
package game.assets;

import org.json.simple.JSONArray;
import org.json.simple.JSONObject;
import org.json.simple.JSONValue;

import java.io.FileNotFoundException;
import java.io.FileReader;

public class StringValues {

    private static JSONObject fileJSON;

    static {
        try {
            fileJSON = (JSONObject) JSONValue.parse(new FileReader( fileName: "src/main/java/game/assets/dialogs.json"));
        } catch (FileNotFoundException ignored) {
        }
    }

    public static String translate(String value) {
        String translatedString = (String) fileJSON.get(value);
        return (translatedString == null) ? value : translatedString;
    }

    public static String translate(String value, int index) {
        String translatedString = (String) ((JSONArray) fileJSON.get(value)).get(index);
        return (translatedString == null) ? value+"."+index : translatedString;
    }

    @Override
    public String toString() { return getClass().getSimpleName(); }
}
```

Figure 27: StringValues Class

This singleton object manages deserialization operations on the dialog.json for fetching any dialog in the game. Such class have methods that are used as such

- translate (String): specify the previously known key
- translate (String, int): specify the previously known key as well as an index (for nested maps)

```java
translate( value: "CONTAINER_DOOR")
```

Figure 28: Translate Invocation

```java
translate( value: "TO_STRING_ROOM", index: 1)
```

Figure 29: Translate Invocation

Section 2.2.2: Containers Package

A package which handles containers and their behaviors; containers are objects that represents as an object in either one of the four walls in each room.

Section 2.2.2.1 Interfaces Package

```
package game.containers.interfaces;

import ...

public interface Container extends java.io.Serializable {

  String getDisplayName();

  Container handleContainerJSONContents(MazeGame mazeGame, JSONObject jsonObject) throws FileNotFoundException;

  void onInteractListener(MazeGame mazeGame, Player player, Contents inventory);
}
```

Figure 30: Container.java

This interface enforces containers to implement these methods, each method act as the following:

- getDisplayName: returns the name of such container in string form (to show it to the user)
- handleContainerJSONContents: each container will receive a child from the JSON file with represents such container in a specific direction, classes that extends lootable should call “getContainerContents” method and assign it to the object contents.
- onInteractListener: called upon user click on the container, inventory is provided as a parameter as well

```java
package game.containers.interfaces;

import ...

public interface ContainerWithLogicCheck {

    ExceptionInfoHolder isLogical(MazeGame mazeGame);
}
```

Figure 31: ContainerWithLogicCheck.java

Triggered immediately when JSON decoding is finished, each container can implement such interface to check whether is JSON deserialized data is logical or not (applied on chests as well as on doors), Non-critical exceptions are raised here.

Section 2.2.2.2 Lootables Package

Contains classes that extends “Lootables” abstract class and explicitly implements “container’ interface.

```java
package game.containers.lootables;

import game.MazeGame;
import game.Player;
import game.containers.interfaces.Container;
import game.loot.*;
import org.json.simple.JSONObject;

import static game.assets.StringValues.*;

public abstract class Lootables implements Container {

    private static final long serialVersionUID = 6592507137261779895L;

    private Contents contents = new Contents.Builder().build();

    public Contents getContents() { return contents; }

    public void setContents(Contents contents) { this.contents = contents; }

    @Override
    public void onInteractListener(MazeGame mazeGame, Player player, Contents inventory) { attemptLoot(player, lootables: this); }

    public static Contents getContainerContents(Object jsonContents) {
        int goldAmount = -1;
        int doorKey = -1;
        int chestKey = -1;
        int flashlightCharge = -1;
        if (((JSONObject) jsonContents).get(translate( value: "JSON_WHICH")).equals(translate( value: "JSON_WHICH_GOLD")))
            goldAmount = jsonToInt(jsonContents, translate( value: "JSON_VALUE"));
        else if (((JSONObject) jsonContents).get(translate( value: "JSON_WHICH")).equals(translate( value: "JSON_WHICH_DOOR_KEY")))
            doorKey = jsonToInt(jsonContents, translate( value: "JSON_VALUE"));
        else if (((JSONObject) jsonContents).get(translate( value: "JSON_WHICH")).equals(translate( value: "JSON_WHICH_CHEST_KEY")))
            chestKey = jsonToInt(jsonContents, translate( value: "JSON_VALUE"));
        else if (((JSONObject) jsonContents).get(translate( value: "JSON_WHICH")).equals(translate( value: "JSON_WHICH_FLASHLIGHT")))
            flashlightCharge = jsonToInt(jsonContents, translate( value: "JSON_VALUE"));

        return new Contents.Builder()
                .withGold(goldAmount)
                .havingDoorKey(doorKey)
                .havingChestKey(chestKey)
                .containingFlashlight(flashlightCharge)
                .build();
    }
```

Figure 32: Lootables.json

Note that this abstract class offers handling the looting process for each class that inherits it, the loot of each lootable container is stored in a “Contents” class object.

The following is an example of a lootable container.

```java
package game.containers.lootables;

import game.MazeGame;
import game.containers.interfaces.Container;
import org.json.simple.JSONObject;

import static game.assets.StringValues.*;

public class Mirror extends Lootables {
  private static final long serialVersionUID = 2064032560879967140L;

  @Override
  public String getDisplayName() { return getClass().getSimpleName(); }

  @Override
  public Container handleContainerJSONContents(MazeGame mazeGame, JSONObject jsonObject) {
    Mirror mirror = new Mirror();
    mirror.setContents(getContainerContents(jsonObject));
    return mirror;
  }

  @Override
  public String toString() {
    return getClass().getSimpleName()+translate( value: "FORMAT_TAB")+getContents().toString();
  }
}
```

Figure 33: Mirror Lootable Container

Section 2.2.2.3 Non-Lootables Package

Contains classes that explicitly implements “container’ interface.

```java
package game.containers.non_lootables;

import static game.assets.StringValues.*;

import game.MazeGame;
import game.Player;
import game.containers.interfaces.Container;
import game.loot.Contents;
import org.json.simple.JSONObject;

public class Wall implements Container {
  private static final long serialVersionUID = 3040187407646395031L;

  @Override
  public String getDisplayName() { return getClass().getSimpleName(); }

  @Override
  public Container handleContainerJSONContents(MazeGame mazeGame, JSONObject jsonObject) { return new Wall(); }

  @Override
  public void onInteractListener(MazeGame mazeGame, Player player, Contents inventory) {
    player.println(translate( value: "UI_WALL"));
  }

  @Override
  public String toString() { return getClass().getSimpleName(); }
}
```

Figure 34: Wall Non-Lootable Container

The main difference between lootables and non-lootables is that non-lootable classes does not have contents “loot”.

Section 2.2.2.4 Container List Class

This class lists all the container classes (lootable and non-lootabele) and its respected value in the JSON file.

It is mainly used as a helper class when fetching from the JSON and assign it as a new class.

```
public class ContainerList {

  static final Map<String, Container> containersList = buildContainersList();

  static final Map<String, ContainerWithLogicCheck> logicCheckList = buildLogicCheckList();

  private ContainerList() {}

  private static Map<String, Container> buildContainersList() {
    Map<String, Container> containersListBuilder = new LinkedHashMap<>();
    containersListBuilder.put(translate( value: "CONTAINER_MERCH"), new Shop());
    containersListBuilder.put(translate( value: "CONTAINER_DOOR"), new Door( to: -1,  isLocked: false));
    containersListBuilder.put(translate( value: "CONTAINER_PAINTING"), new Painting());
    containersListBuilder.put(translate( value: "CONTAINER_MIRROR"), new Mirror());
    containersListBuilder.put(translate( value: "CONTAINER_CHEST"), new Chest());
    containersListBuilder.put(translate( value: "CONTAINER_WALL"), new Wall());
    return containersListBuilder;
  }

  private static Map<String, ContainerWithLogicCheck> buildLogicCheckList() {
    Map<String, ContainerWithLogicCheck> logicCheckListBuilder = new LinkedHashMap<>();
    logicCheckListBuilder.put(
            translate( value: "CONTAINER_DOOR"), (ContainerWithLogicCheck) containersList.get(translate( value: "CONTAINER_DOOR")));
    logicCheckListBuilder.put(
            translate( value: "CONTAINER_CHEST"), (ContainerWithLogicCheck) containersList.get(translate( value: "CONTAINER_CHEST")));
    return logicCheckListBuilder;
  }

  public static Map<String, Container> getContainersList() {
    return containersList;
  }

  public static Map<String, ContainerWithLogicCheck> getLogicCheckList() {
    return logicCheckList;
  }

  @Override
  public String toString() {
    return getClass().getSimpleName();
  }
}
```

Figure 35: Container List Class

Section 2.2.2.4 Room Class

Used as a placeholder for containers in each direction (east, north, west, south), stores whether such room is dark or not.

```
public class Room<T extends Container> implements java.io.Serializable {
  private static final long serialVersionUID = 7295462279118008713L;

  private final boolean isDark;
  private final T east;
  private final T north;
  private final T west;
  private final T south;

  public Room(boolean isDark, T east, T north, T west, T south) {
    this.isDark = isDark;
    this.east = east;
    this.north = north;
    this.west = west;
    this.south = south;
  }

  public boolean isDark() {
    return isDark;
  }

  public T getEast() {
    return east;
  }


  public T getNorth() {
    return north;
  }


  public T getWest() {
    return west;
  }


  public T getSouth() {
    return south;
  }
```

Figure 36: Room Class

Section 2.2.3: Exceptions Package

Includes exception classes and placeholders.

Section 2.2.3.1 Exception Info Holder

A class which holds information about the raised exception, whether if it was a success or not, as well as mentioning the reason.

```java
package game.exceptions;

import static game.assets.StringValues.*;

public class ExceptionInfoHolder {
    private final boolean isSuccess;
    private final String reason;

    public ExceptionInfoHolder(boolean isSuccess, String reason) {
        this.isSuccess = isSuccess;
        this.reason = reason;
    }

    public boolean isSuccess() { return isSuccess; }

    public String getReason() { return reason; }

    @Override
    public String toString() {
        return getClass().getSimpleName()
                + translate( value: "TO_STRING_EXCEPTION_INFO_HOLDER", index: 0)
                + isSuccess
                + translate( value: "TO_STRING_EXCEPTION_INFO_HOLDER", index: 1)
                + reason;
    }
}
```

Figure 37: Exception Info Holder

Section 2.2.3.2 Illogical Mapping Exception Class

An exception class which indicates -if raised- that there is a non-logical deserialized data may cause the game to be either unwinnable or cause an instant win.

Refer to section one for more details on how to create a proper JSON map file.

```
package game.exceptions;

import static game.IoHandler.printlnSys;
import static game.assets.StringValues.*;

public class IllogicalMapping extends Exception {
  public IllogicalMapping(String s) {
    super(s);
    printlnSys(s);
  }

  @Override
  public String toString() {
    return getClass().getSimpleName()
        + translate( value: "TO_STRING_ILLOGICAL_MAPPING");
  }
}
```

Figure 38: Illogical Mapping Exception Class

Section 2.2.4: Loot Package

A package which contains the following

- Contents: Contains all the loot classes, employs the builder design pattern.
- Chest keys: Responsible for opening chest containers (contains a number denoting a respected chest).
- Door keys: Responsible for opening door containers (contains a number denoting a respected door).
- Flashlight: Simply contains an integer which decrements by five when exploring a dark room
- Gold: Used for trading with the merchant, contributes to the winning conditions.

Section 2.2.4.1 Contents Class

Note that both containers and players have contents object as a variable.

```java
public class Contents implements java.io.Serializable {
  private static final long serialVersionUID = 10957757670735087 78L;

  private final Gold gold;
  private final FlashLight flashlight;
  private final List<DoorKey> doorKeys;
  private final List<ChestKey> chestKeys;

  public static class Builder {
    private Gold gold = new Gold( amount: -1);
    private FlashLight flashlight = new FlashLight( charge: 0);
    private final List<DoorKey> doorKeys = new ArrayList<>();
    private final List<ChestKey> chestKeys = new ArrayList<>();

    public Builder withGold(int amount) {
      if (amount > 0) this.gold = new Gold(amount);
      return this;
    }

    public Builder containingFlashlight(int charge) {
      if (charge > 0) this.flashlight = new FlashLight(charge);
      return this;
    }

    public Builder havingDoorKey(int doorKey) {
      if (doorKey > 0) this.doorKeys.add(new DoorKey(doorKey));
      return this;
    }

    public Builder havingChestKey(int chestKey) {
      if (chestKey > 0) this.chestKeys.add(new ChestKey(chestKey));
      return this;
    }

    public Contents build() { return new Contents( builder: this); }
  }

  private Contents(Builder builder) {
    this.gold = builder.gold;
    this.flashlight = builder.flashlight;
    this.doorKeys = builder.doorKeys;
    this.chestKeys = builder.chestKeys;
  }
```

Figure 39: Contents Class

```
return new Contents.Builder()
        .withGold(goldAmount)
        .havingDoorKey(doorKey)
        .havingChestKey(chestKey)
        .containingFlashlight(flashlightCharge)
        .build();
```

Figure 40: Contents Builder Class Usage

```
private final Contents inventory = new Contents.Builder().build();
```

Figure 41: Contents Object with Empty Loot

Section 2.2.4.2 Chest Key Class

Contains an integer which specifies which key refers to which chest, a chest will not open unless its key is within the player's inventory.

```java
package game.loot;

import java.util.Objects;

import static game.assets.StringValues.*;

public class ChestKey implements java.io.Serializable {
  private static final long serialVersionUID = 2409548198138641500L;

  final int which;

  public ChestKey(int which) { this.which = which; }

  public int getWhich() { return which; }

  @Override
  public boolean equals(Object o) {
    if (o instanceof ChestKey) {
      return which == ((ChestKey) o).which;
    } else return false;
  }

  @Override
  public int hashCode() { return Objects.hash(which); }

  @Override
  public String toString() {
    return getClass().getSimpleName() + translate( value: "TO_STRING_CHEST_KEY") + which;
  }
}
```

Figure 42: Chest Key Class

Section 2.2.4.3 Door Key Class

Contains an integer which specifies which key refers to which door, a chest will not open unless its key is within the player's inventory (or the door was not locked in the first place), once the door is open, other players will be able to open the door without the key.

```
package game.loot;

import java.util.Objects;

import static game.assets.StringValues.*;

public class DoorKey implements java.io.Serializable {
  private static final long serialVersionUID = 3041654785548004192L;

  final int which;

  public DoorKey(int which) { this.which = which; }

  public int getWhich() { return which; }

  @Override
  public boolean equals(Object o){
    if(o instanceof DoorKey){
      return this.which==((DoorKey) o).which;
    } else
      return false;
  }

  @Override
  public int hashCode() { return Objects.hash(which); }

  @Override
  public String toString() {
    return getClass().getSimpleName()+translate( value: "TO_STRING_DOOR_KEY")+which;
  }
}
```

Figure 43: Door Key Class

Section 2.2.4.4 Flashlight Class

Specifies the charge, as an AtomicInteger; to handle concurrency issues. The charge is deducted by 5% whenever the player does the following:

- Enters a dark room
- Interacts to a container in a dark room

```java
package game.loot;

import java.util.concurrent.atomic.AtomicInteger;

import static game.assets.StringValues.*;

public class FlashLight implements java.io.Serializable {
    private static final long serialVersionUID = 2314863596349842107L;

    AtomicInteger charge;

    public FlashLight(int charge) { this.charge = new AtomicInteger(charge); }

    public int getCharge() { return charge.get(); }

    public void addCharge(int charge) {
        this.charge = new AtomicInteger( initialValue: this.charge.get() + charge);
        if (this.charge.get() < 0) this.charge = new AtomicInteger( initialValue: 0);
    }

    @Override
    public String toString() { return getClass().getSimpleName() + translate( value: "TO_STRING_FLASHLIGHT") + charge; }
}
```

Figure 44: Flashlight Class

Section 2.2.4.5 Gold Class

Specifies the gold amount, as an AtomicInteger; to handle concurrency issues. Gold it used in the following scenarios:

- Trade with merchants
- Reach victory conditions
- Defeat players who have less gold

```java
package game.loot;

import java.util.concurrent.atomic.AtomicInteger;

import static game.assets.StringValues.*;

public class Gold implements java.io.Serializable {
    private static final long serialVersionUID = 6946382482265401424L;

    AtomicInteger amount;

    public Gold(int amount) { this.amount = new AtomicInteger(amount); }

    public int getAmount() { return amount.get(); }

    public void addAmount(int amount) {
        this.amount = new AtomicInteger( initialValue: this.amount.get() + amount);
        if (this.amount.get() < 0) this.amount = new AtomicInteger( initialValue: 0);
    }

    public void setAmount(int amount) {
        this.amount = new AtomicInteger(amount);
    }

    @Override
    public String toString() { return getClass().getSimpleName() + translate( value: "TO_STRING_GOLD") + amount; }
}
```

Figure 45: Gold Class

Section 2.2.5: Controls Class

Used to define the value of buttons and binds it together.

```java
package game;

import java.util.Random;

public class Controls {
    private static final char FORWARD = 'W';
    private static final char RIGHT = 'D';
    private static final char LEFT = 'A';
    private static final char BACKWARD = 'S';
    private static final char STATUS = 'T';
    private static final char EXIT = 'E';
    private static final char ENGAGE = 'K';
    private static final char ROCK = '1';
    private static final char PAPER = '2';
    private static final char SCISSORS = '3';


    private Controls() {
    }

    public static char getForwardKey() { return FORWARD; }

    public static char getRightKey() { return RIGHT; }

    public static char getLeftKey() { return LEFT; }

    public static char getBackwardKey() { return BACKWARD; }

    public static char getStatusKey() { return STATUS; }

    public static char getExitKey() { return EXIT; }

    public static char getEngageKey() { return ENGAGE; }

    public static char getRockKey() { return ROCK; }

    public static char getPaperKey() { return PAPER; }

    public static char getScissorsKey() { return SCISSORS; }

    public static char getRandomTieBreakerChoice() { return (new Random().nextInt( bound: 4) + " ").charAt(0); }

    @Override
    public String toString() { return getClass().getSimpleName(); }
}
```

Figure 46: Controls Class

Section 2.2.6: Countdown Class

Employed in the remaining time of a single game as a thread, the servlet will get the remaining time and sets to refresh the page after the interval finishes, the JSP will run a JS function to stimulate the countdown timer without the need to refresh every second.

```
public class Countdown implements Runnable {
    Timer timer = new Timer();
    private MazeGame mazeGame;
    private int startingCountdown;
    private int remaining;
    private boolean isStarted = false;

    public Countdown(MazeGame mazeGame) { this.mazeGame = mazeGame; }

    public boolean isStarted() { return isStarted; }

    public int getRemaining() { return remaining; }

    public void setStartingCountdown(int startingCountdown) {
        if (this.startingCountdown == 0) this.startingCountdown = startingCountdown;
        else printlnSys(translate( value: "TIMER_ALREADY_SET"));
        remaining = startingCountdown;
    }

    private void startTimer() {
        isStarted = true;
        timer.scheduleAtFixedRate(
                () -> { updateRemaining(); },
                 delay: 1000,
                 period: 1000);
    }

    private void updateRemaining() {
        if (remaining < 0) {
            GamesList.gamesList.remove(mazeGame);
            for (Player player : mazeGame.getPlayers().keySet())
                player.setMazeGame(null);
            mazeGame = null;
            timer.cancel();
        }
        --remaining;
    }

    @Override
    public void run() { startTimer(); }
```

Figure 47: Countdown Class

Section 2.2.7: Games List Class

Stores a list of current games (started or not), if the countdown timer reaches 0, the game with the thread assigned to it will be deleted.

```
public class GamesList {
    private static final List<MazeGame> GAMES_LIST = new ArrayList<>();

    private GamesList() {
    }

    public static int hostGame() {
        MazeGame mazeGame = new MazeGame();
        GAMES_LIST.add(mazeGame);
        return GAMES_LIST.size() - 1;
    }

    public static Map<String, String> joinGame() {
        LinkedHashMap<String, String> games = new LinkedHashMap<>();
        for (MazeGame mazeGame : GAMES_LIST)
            if (!mazeGame.isStarted())
                games.put(mazeGame.getId(),
                        mazeGame.getPlayers().entrySet().iterator().next().getKey().getName() + "'s room - "
                                + mazeGame.getPlayers().size() + "/" + mazeGame.getCapacity() + " player(s) - "
                                + mazeGame.getMap() + " map");
        return games;
    }

    public static List<String> getMaps() {
        ArrayList<String> mapsList = new ArrayList<>();

        File folder = new File(translate( value: "MAP_DIRECTORY"));
        File[] listOfFiles = folder.listFiles();

        if (listOfFiles == null) return mapsList;

        for (File listOfFile : listOfFiles)
            if (listOfFile.isFile())
                mapsList.add(listOfFile.getName().substring(0, listOfFile.getName().indexOf('.')));

        return mapsList;
    }

    public static List<MazeGame> getGamesList() {
        return GAMES_LIST;
    }
}
```

Figure 48: Games List Class

Section 2.2.8: IO Handler Class

Used to manage logging operation (writes into Log.log file).

```java
package game;

import org.apache.log4j.Logger;

public class IoHandler {

  private static final Logger logger = Logger.getLogger(IoHandler.class.getName());

  private static String batch = "";

  private IoHandler() {}

  public static void printlnSys(String string) {
    string = batch + string;
    while (string.contains("\n")) {
      if (string.contains("\n\n")) break;

      String line = string.substring(0, string.indexOf("\n"));
      logger.debug(line);
      string = string.substring(string.indexOf("\n") + 1);
      batch = "";
    }
    logger.debug(string);
    batch = "";
  }

  public static void printlnSys() {
    logger.debug(batch);
    batch = "";
  }

  public static void printSys(String string) { batch += string; }

  @Override
  public String toString() { return getClass().getSimpleName(); }
}
```

Figure 49: IO Handler Class

Section 2.2.9: Maze Game Class

Manages operations related to a specific maze game instance, such as JSON decoding and populating the map with containers, as well as finding the maximum capacity of the game; by counting lit rooms. This class is also responsible for starting up the counter (once the host launches the game).

```
public void decodeJSON(String chosenMap) throws FileNotFoundException, IllogicalMapping {
    map = chosenMap;
    if (isDecoded)
        return;

    isDecoded = true;

    jsonDirectory = translate( value: "JSON_DIR") + chosenMap + ".json";

    JSONObject fileJSON = (JSONObject) JSONValue.parse(new FileReader(jsonDirectory));

    JSONArray roomsJSON = (JSONArray) fileJSON.get(translate( value: "JSON_ROOMS"));

    if (countdown.getRemaining() == 0)
        countdown.setStartingCountdown(jsonToInt(fileJSON, translate( value: "JSON_REMAINING_TIME")));
    initialGold = jsonToInt(fileJSON, translate( value: "JSON_INITIAL_GOLD"));
    setGoldThresholdObjective(jsonToInt(fileJSON, translate( value: "JSON_GOLD_THRESHOLD")));

    for (Object room : roomsJSON) {
        boolean isDark = (boolean) ((JSONObject) room).get(translate( value: "JSON_IS_DARK"));

        String eastContainerType = getContainerType(room, translate( value: "JSON_EAST"));
        String northContainerType = getContainerType(room, translate( value: "JSON_NORTH"));
        String westContainerType = getContainerType(room, translate( value: "JSON_WEST"));
        String southContainerType = getContainerType(room, translate( value: "JSON_SOUTH"));

        getRooms()
                .add(
                        new Room<>(
                                isDark,
                                getContainer(((JSONObject) room).get(translate( value: "JSON_EAST")), eastContainerType),
                                getContainer(((JSONObject) room).get(translate( value: "JSON_NORTH")), northContainerType),
                                getContainer(((JSONObject) room).get(translate( value: "JSON_WEST")), westContainerType),
                                getContainer(((JSONObject) room).get(translate( value: "JSON_SOUTH")), southContainerType)));
    }
    checkMappingLogic();
}

public String getContainerType(Object room, String json) {
    JSONObject child = (JSONObject) ((JSONObject) room).get(json);
    return (String) child.get(translate( value: "JSON_TYPE"));
}

public int jsonToInt(Object jsonObject, String location) {
    return ((Long) ((JSONObject) jsonObject).get(location)).intValue();
```

Figure 50: Maze Game Class

Section 2.2.10: Player Class

Manages operations related to a specific player, such as storing the player's acquired loot, as well as handling interactions with the map/opponents. The main responsibilities of the class are thus:

- Choosing a random spawn point
- Allocating the startup gold
- Handling moving operations
- Handling looting operations
- Handling fighting scenarios
- Drawing the map
- Print the status
- Killing a player
    - The player himself (upon losing or leaving the game)
    - The opponent when he wins

Note that the player cannot interact or loot any container unless he defeats an opponent (if present in same room).

```
private boolean isRoomClearOfPlayers(Room<Container> randomLocation) {
    for (Player player : mazeGame.getPlayers().keySet())
        if (player.getLocation() != null && player.getLocation().equals(randomLocation))
            return false;
    return true;
}

public void move(char inputCommand) {
    command = Character.toUpperCase(inputCommand);
    if (Boolean.TRUE.equals(mazeGame.getPlayers().get(this))) return;
    if (command == getStatusKey()) printStatus(mazeGame);
    else if (command == getExitKey()) kill( victim: this, getNearestContainer().getContents());
    else if (command == getEngageKey()) engagePlayer();
    else if (command == getRockKey()) {
        tieBreakerChoice = 1;
        engagePlayer();
    } else if (command == getPaperKey()) {
        tieBreakerChoice = 2;
        engagePlayer();
    } else if (command == getScissorsKey()) {
        tieBreakerChoice = 3;
        engagePlayer();
    } else if (isClearForManeuver( player: this, getLocation())) handleManeuver();
    println(translate( value: "UI_AWAITING_COMMAND"));
}

private void engagePlayer() {
    Player opponent = null;

    for (Player player : mazeGame.getPlayers().keySet())
        if (getLocation() == player.getLocation() && player != this)
            opponent = player;

    if (opponent != null) {
        if (getInventory().getGold().getAmount() < opponent.getInventory().getGold().getAmount())
            kill( victim: this, opponent.getInventory());
        else if (getInventory().getGold().getAmount() > opponent.getInventory().getGold().getAmount()) {
            kill(opponent, this.getInventory());
            println(opponent.getName() + translate( value: "KILLED"));
        } else
            triggerTieBreaker(opponent);

    }

}
```

Figure 51: Player Class

Section 2.2.11: Store Class

Stores store items inside for each maze game, stores in a single maze game share the same items.

Store items are represented as a HashMap (the key is the Contents, and the value is the price).

```
package game;

import game.loot.Contents;
import java.util.LinkedHashMap;
import java.util.Map;

import static game.assets.StringValues.*;

public class Store {
  private final LinkedHashMap<Contents, Integer> storeItems = new LinkedHashMap<>();

  public Map<Contents, Integer> getStoreItems() {
    return storeItems;
  }

  public String getStoreItemsString() {
    StringBuilder items = new StringBuilder();
    for (Map.Entry<Contents, Integer> contents : storeItems.entrySet())
      items
          .append(contents.getKey().toString())
          .append(translate( value: "TO_STRING_STORE", index: 0))
          .append(contents.getValue())
          .append(translate( value: "TO_STRING_STORE", index: 1));
    return Store.class.getSimpleName() + translate( value: "TO_STRING_STORE", index: 2) + items;
  }

  @Override
  public String toString() { return getStoreItemsString(); }
}
```

Figure 52: Store Class

Section 3: Design Patterns Compliance

*The code also satisfies Google's styling guide implicitly since google-java-format is installed; by pressing Ctrl+Alt+L, IntelliJ automatically reformats the code by google formatting xml file rules

Practice / Design Pattern	Description	Compliant?	Reason
Builders	Used in a situation where there are many constructors with a chance of extending	Compliant	A container may have a variety of contents (loot)
Singleton	Deprive the opportunity of creating an object	Compliant	Having static variables and methods with a private default constructor is a form of singleton pattern
Enforce noninstantiability with a private constructor	Deprive the opportunity of creating an object	Compliant	Many classes have private constructor with static variables and methods
Avoid finalizers	Uncaught exceptions inside a finalizer will not print a warning, it will also cause severe performance penalty	Compliant	Finalizers are not implemented in any package
Always override hashCode when you override equals	Help the class to function properly with all hash-based collections (mainly for objects distinction)	Compliant	All classes which override the equals method also overrides the hashCode method
Always override toString	Makes the class much more pleasant to read	Compliant	All classes (non-abstract) have the toString method overridden (returns all of the interesting information contained in the object)
Minimize the accessibility of classes and members	Hides all of its implementation details	Compliant	Some stringValues's variables use unmodifiableList rather than a mere list, making it immutable
In public classes, use accessor methods, not public fields	Implementing accessor methods (getters) and mutators (setters)	Compliant	All variables are private and can only be accessed with a getter/ setter (if saw fit)
Do not use raw types in new code	Use of raw types lose safety and expressiveness of generics (compiler will not have enough type information to perform all type checks necessary to ensure type safety)	Compliant	All generics are parameterized
Eliminate unchecked warnings	Suppressing warning hides flaws in the code, which will be harder to troubleshoot if an issue arises	Compliant	There are no suppressed warnings used in the project
Prefer lists to arrays	Lists provides a versatile way to manipulate data, use arrays over lists only when the length is constant	Compliant	Arrays are never used, since there is no preferred implementation for it (excluding getting a list of maps)
Consistently use the Override annotation	Use the Override annotation on every method declaration that you believe to override a super class declaration.	Compliant	Ever overridden method has the annotation
Use varargs judiciously	varargs methods are a convenient way to define methods that require a variable number of arguments, but they should not be overused	Compliant	Batch controls can receive an indefinite number of controls
Return empty arrays or collections, not nulls	There is no reason ever to return null from an array- or collection-valued method instead of returning an empty array or collection	Compliant	Null values are not even used in this project
Prefer for-each loops to traditional for loops	For-each is easier to read and have relatively less coding	Compliant	For-each is implemented, unless a counter is needed, then the for loop would be the better choice
Know and use libraries	Using libraries ensures that the code is written have high standards since experts wrote it	Compliant	Log4j is implemented here, also java.util and java.io
Prefer primitive types to boxed primitives	Primitives are more space and time efficient	Compliant	Primitives are always used when applicable, boxed primitives are used in generics
Avoid Strings where other types are more appropriate	Strings tend to be slower, less flexible and more error-prone	Compliant	String data type is only used to represent data and for serializing

Practice / Design Pattern	Description	Compliant?	Reason
Beware the performance of string concatenation	Using the string concatenation operator repeatedly to concatenate n strings requires time quadratic in n.	Compliant	StringBuilder is used when there are too many concatenations
Adhere to generally accepted naming conventions	Follow typographical and grammatical naming conventions	Compliant	All variable naming complies to those conventions
Use exceptions only for exceptional conditions	Exceptions are used when unwanted events transpire	Compliant	Exception can be thrown when there is a serialization error, or it could be as simple as the timer runs out
Use checked exceptions for recoverable conditions and runtime exceptions for programming errors	checked exceptions: for conditions from which the caller can reasonably be expected to recover unchecked exceptions: should not be caught.	Compliant	Most of the checked exceptions are recoverable, unless if it will impair the rest of the code, fail-fast approach is imposed.
Favor the use of standard exceptions	Easier to read and use because it matches established conventions with which programmers are already familiar	Compliant	User defined exceptions are only made there are no standard exception that fulfills the situation's needs
Do not ignore exceptions	Do not let catch blocks empty.	Compliant	All catch blocks have their stack traces logged and printed out
Consider using a custom serialized form	Do not accept the default serialized form without first considering whether it is appropriate.	Compliant	An explicit serial version UID in every serializable class is declared
Solid Principles Compliance			
Single Responsibility Principle	A class should have one, and only one, reason to change.	Partial Compliance	Having only one job for each class may defeat the purpose of encapsulation, there should be a "sweat spot" for decomposing the jobs into sub-jobs
Open Closed Principle	You should be able to extend a classes behavior, without modifying it.	Compliant	Some components, such as containers, can be extended without modifying the base code
Liskov Substitution Principle	Derived classes must be substitutable for their base classes.	Compliant	The base class matches with the specifications of the parent class (a door is a container, while a chest is a lootable container)
Interface Segregation Principle	Make fine grained interfaces that are client specific.	Compliant	Each container must implement three methods, however, there are some optional methods that the user can choose to implement or not
Dependency Inversion Principle	Depend on abstractions, not on concretions.	Partial Compliance	Same as the O in the SOLID, the containers have a layer of abstraction with its dependances, but that is not always the case on all of the classes
Sonar List Compliance (All SonarLint rules are respected)			
Cognitive Complexity	Each method should have at most 15 control statements in each method	Partial Compliance	Large method now comprises into smaller, logical methods. However, implementing such behavior on servlet overridden methods may result into a negative effect; as it will cause new methods to have too many passed methods.
Limit Break/Continue Statements to One Per Loop	Doing so improves code tracking	Compliant	All loops contain at most one break/continue statement
Replace the use of System.out or System.err by a logger	Standard outputs should not be used directly to log anything	Compliant	All output to the console is implicitly logged into a log file
Provide the parametrized type for this generic.	Generic wildcard types should not be used in return types	Compliant	There are no wildcards implemented into the program
Use indentation to denote the code conditionally executed by this "if"	A conditionally executed single line/block should be denoted by indentation	Compliant	Indentation is Implemented

Section 4: Extension Scenario

Suppose that we need a new container called “thief”, which steals gold from the player when interacted.

Requirements:

1. Fill your desired map with the locations where the thief happens to be there, for simplicity, we will be adding two rooms for the sample map, the first one contains one thief, and the other contains two thieves, each thief contains a “deduct” attribute (specifies the amount of stolen gold) which will be decoded later, remember to adhere to the logical rules when creating any map since its very strict.

2. Create a new container class called thief in the subdirectory of “non_lootables” (since the thief should not contain any loot for the player)

```
package game.containers.non_lootables;

public class Thief {
}
```

3. In order for the class to adhere to the container rules, we must implement the container interface (for this example, we will not implement the optional interfaces)

```
public class Thief implements Container {
    @Override
    public String getDisplayName() {
        return null;
    }

    @Override
    public Container handleContainerJSONContents(JSONObject jsonObject) throws FileNotFoundException {
        return null;
    }

    @Override
    public void onInteractListener(Contents inventory) {
    }
}
```

4. Fill the methods in the following logic:
    a. **The first method returns a string which will be shown during the map drawing.** Make the display name return “Chest”; so that the user is tricked into thinking that the container is a chest filled with loot.

    b. **The second method handles the JSON decoding that you previously made, for convenience, the method already gives you the JSON file which contains the child where you specified as the thief.** You should read the JSON and create a new object here

    c. **The third method gets invoked whenever the user interacts with the container during the game.** The stealing process should occur here, you can have any method from the program at your disposal, even though this will result in highly dependent classes, it will improve the flexibility of the containers, such that the container can control anything, for example, depleting the flashlight

```java
public class Thief implements Container {
  boolean isStolen = false;
  int stealingAmount;

  public Thief(int stealingAmount) {
    this.stealingAmount = stealingAmount;
  }

  @Override
  public String getDisplayName() {
    return "Chest";
  }

  @Override
  public Container handleContainerJSONContents(JSONObject jsonObject) {
    return new Thief(((Long) jsonObject.get("deduct")).intValue());
  }

  @Override
  public void onInteractListener(Contents inventory) {
    if (!isStolen) {
      int startingGold = inventory.getGold().getAmount();
      int stolenGold = min(startingGold, stealingAmount);
      if (stolenGold == 0) println("The thief pitied me");
      else println("A thief stole from me " + stolenGold + " gold");
      inventory.getGold().setAmount(startingGold - stolenGold);
      isStolen = true;
    } else println("Speaking of fooling me twice :|");
  }
}
```

5. Even though the class is fully operational, the program must be able to recognize and call it automatically; head to game → containers → ContainerList and add to the “buildContainersList” hash map the following:
    a. A string which denotes the JSON type of container
    b. An object instance of such container

```java
private static Map<String, Container> buildContainersList() {
  Map<String, Container> containersListBuilder = new LinkedHashMap<>();
  containersListBuilder.put(CONTAINER_MERCH, new Shop());
  containersListBuilder.put(CONTAINER_DOOR, new Door( to: -1,  isLocked: false));
  containersListBuilder.put(CONTAINER_PAINTING, new Painting());
  containersListBuilder.put(CONTAINER_MIRROR, new Mirror());
  containersListBuilder.put(CONTAINER_CHEST, new Chest());
  containersListBuilder.put(CONTAINER_WALL, new Wall());
  containersListBuilder.put("Thief", new Thief( stealingAmount: -1));
  return containersListBuilder;
}
```

Note that if you have implemented any optional interfaces, you will have to add into the other hash maps as well.

Observe how the program communicates with the “Thief” class functionality, note that the pseudo chest is here while there is a thief underlying it, the same would apply on the JSP extension; however, you need to add an image with the same name of the “getDisplayName” (in this example, we will not be adding an image since the chest image file is already embedded into the resources).

```
16:11:13 | ----------------------------------------
16:11:13 |       ********Door********
16:11:13 |       *                  *
16:11:13 |       *                  *
16:11:13 |      Chest               Shop
16:11:13 |       *                  *
16:11:13 |       *                  *
16:11:13 |       *******Mirror*******
16:11:13 | ----------------------------------------
16:11:13 | I have 60 seconds to collect 1000 gold, better get to it.
16:11:13 | Awaiting command...
t
16:11:14 |
16:11:14 | Status
16:11:14 | ----------------------------------------
16:11:14 | Gold: 800/1000
16:11:14 | ----------------------------------------
16:11:14 | Remaining time: 60 seconds
16:11:14 | ----------------------------------------
16:11:14 | Awaiting command...
16:11:14 | Only one minute left, I still need 200 gold.
a
16:11:15 | A thief stole from me 100 gold
16:11:15 | Awaiting command...
a
16:11:15 | Speaking of fooling me twice :|
16:11:15 | Awaiting command...
t
16:11:16 |
16:11:16 | Status
16:11:16 | ----------------------------------------
16:11:16 | Gold: 700/1000
16:11:16 | ----------------------------------------
16:11:16 | Remaining time: 58 seconds
16:11:16 | ----------------------------------------
16:11:16 | Awaiting command...
```

Section 5: Further Notes

There are a few notes that should be covered before diving into the code.

- Even though the container classes follow the open–closed principle, however, the same thing cannot be said for loot classes, since it is tightly coupled to the logic of the game. For instance, if we added a new loot such as silver, we would have to create a whole new means on how holding such loot would affect the gameplay, i.e., it does not contribute to the winning conditions, but it can be traded, or you can convert it into gold, etc. Also, this will result in interfering with the containers, so that they can interact with such loot.

- All string values have been migrated into the “Dialogs.json” file and can be used by calling “translate” method from the “StringValues”, this should ease changing any dialog/exception messages.

- Any major events and exceptions are already being logged into the “Log.log” file located in the “assets” directory.

- Any loot that has been inserted into the JSON with an invalid value (less than zero) is omitted.

- Containers may throw an exception if you do not comply with their guidelines.

- If you encountered an error code 3001, it may be because of the following reasons:
    - The JSON syntax is invalid.
    - The program received an unexpected datatype, for example, found string while expecting an integer.
    - A new class is created but not added into the hash map.
    - The JSON keys are undefined.
    - A JSON key was expected, but not found.

  The reason and the line number to troubleshoot the problem will be at your disposal

```
17:17:58 | Error decoding the JSON file, ensure that the structure is compliant (error code 3001)
17:17:58 | ----------------------------------------
17:17:58 | Cannot invoke "java.lang.Boolean.booleanValue()" because the return value of "org.json.simple.JSONObject.get(Object)" is null
17:17:58 | Line number: 165
```

- The program does not follow all of the requested requirements in the assignment, this includes:

- **The UI is based on string commands:** it is highly improbable to have a game with commands in a form of string rather than a single character (pressing “W” is far more convenient than entering “forward”).

- **Selling to the trader:** doing so could break the game, since when selling an imperative loot such as a key, the user will never be able to open the container anymore.

- **Silhouette:** you can just print something when interacting with the mirror in the class’s “onInteractListener” method.

```
@Override
public void onInteractListener(Contents inventory) {
  super.onInteractListener(inventory);
  println("I can see myself :)");
}
```

```
17:43:49 | ----------------------------------------
17:43:49 |       ********Door********
17:43:49 |       *                  *
17:43:49 |       *                  *
17:43:49 |      Door               Door
17:43:49 |       *                  *
17:43:49 |       *                  *
17:43:49 |       *******Mirror*******
17:43:49 | ----------------------------------------
17:43:49 | Awaiting command...
$
17:43:50 | Found key for chest/s number
17:43:50 | 4
17:43:50 | I can see myself :)
17:43:50 | Awaiting command...
```

- **Having an exit door as a winning condition:** changed into collecting gold until a threshold is reached.

- Each player is given a random generated ID, this is considered as a bad practice since abusive users can exploit this weakness and start spamming host/join requests. The solution would be to add the player’s IP address as the ID, so if a player keeps hosting repetitively, they will only create on instance of the game based on their own IP address. The IP address can be fetched by calling “request.getRemoteAddr()” in the servlet.

- Most pages are being auto-refreshed every five second interval for fetching new updated data from the Java classes, this is also considered as a bad practice, the optimal solution would be by utilizing AJAX queries; hence, avoiding refreshing the page.

- **Concurrency:**
  Since there was a required timer, it is in our best interest to utilize threads. There is a timer thread for each game object.

  Also, introducing multiplayer functionality will draw many concerns, the following is a list of the main problems and its mitigation:

Concurrency Issue	Description	Mitigation
Hosting a game	Hosting a game will implicitly decode the JSON into the containers, and decoding the JSON will create container objects, chests and doors are given by an id that are incremented when each object is create; hence each JSON decoding must be impose the “await” pattern	Make “decodeJSON” method synchronized
Buying from the trader	Players may buy the same item at the same time	Make the transaction method synchronized, and deny the last player from buying it
Distributing the defeated player’s assets	When a player is defeated all players receive a portion of his gold	Introduce atomic variable types to the gold

- **Used data structures:**
  - LinkedHashMap “HashMap”: Needed for a key/value pair (loot and price)
  - HashSet “Set”: ordered, nonrepeatable (comparing chest keys with chests id)
  - ArrayList “List”: A versatile approach to store elements than an array (store keys in the inventory)