**Zuul Station – PPA Assignment 2 Report**

Harry Tennent: K1890723

**Zuul Station**

Zuul Station is a text-based adventure game. The setting of the game is a crashed space station, hundreds of thousands of miles away from Earth. In the crash, the oxygen supply to the station was cut off and now there is a limited amount of oxygen left until the station runs out. The aim of the game is to find some codes for the escape pods and take them to the escape pod room. Once this is done, the player has won. The player has 3 minutes to do this before the oxygen for the station runs out. However, there are items in the game that can add different amounts of time to the game. The player has a backpack with a limited capacity and can pick items up. The player can use items, drop items, or they can give the items to other characters. These characters are randomly created and move through the game like the player. There are 10 rooms in the game: a living quarters, main hall, laboratory, reactor, medical bay, kitchen, store room, gym, escape pod room, and a transporter. The player can walk throughout all of these rooms freely. The transporter room transports the player to a random room in the space station. The player has to find the escape pod room, along with the codes, to win.

**Base Tasks:**

**The game has several locations/rooms.**

The ‘zuul-better’ project already had a number of rooms implemented in it as well as a separate Room class. I changed the rooms pre-existing in the game to suit the theme of the game I was making. There are 10 rooms which were: a living quarters, main hall, laboratory, reactor, medical bay, kitchen, store room, gym, escape pod room, and a transporter room. The only room that was different from the rest was the transporter room which I implemented as part of one of the challenge tasks.

The player can walk through the locations.

This was already implemented in the code I was given and I just adding exits to each of the rooms that I added so that the player could move between rooms.

**There are items in some rooms. Every room can hold any number of items. Some items can be picked up by the player, others can’t.**

I created a new class to represent an item. Every item has a name, a weight, a boolean value for whether it can be picked it up, and a bonus time associated with using the item. All of this data was stored in a hash map where the key was a string of the item name, and the value was an array list. Within the array list, I stored the weight, whether it can be picked up, and its bonus time. The values were always in the same order which made it easy to retrieve specific values for a particular item when required. Initially, I had 2 separate classes for handling items. One stored all of the data about each of the different types of items and another class representing a single item that was initialised by that first class. However, I found I was duplicating many methods and it made the implementation of other methods complex, so I decided to simplify it to one class. I have made methods to set random values to the fields of the item class, so the Room can create many random Item objects and I made methods to retrieve data about specific items. Items in a room are stored in a hash map stored in the Room object, so the object can manipulate the items (e.g. when they are picked up) and can get information on the items. Items in the room can be picked up by the player and added to their backpack, provided they have enough space in their backpack and that the item’s field about whether it can be picked up is set to true.

**The player can carry some items with him. Every item has a weight. The player can carry items only up to a certain total weight.**

To implement this, I created a class called Player. In this class I created a “backpack” which was an array list of strings. Accessor methods allowed the Game class to get individual items from the backpack at an index or get them by name. There are also mutator methods so items can be added to the player’s backpack, provided that there is enough space available. To check the space in the backpack, I made a constant “MAX\_WEIGHT” which is initialised to 100. Then, when items are added to the backpack, I check the current backpack weight against this constant to make sure there is enough space. If there is not enough space in the backpack, then the player cannot pick up the item. When the player drops or gives an item to a character, the current weight is reduced.

**The player can win. There has to be some situation that is recognised as the end of the game where the player is informed that they have won.**

For the player to win, they must find the codes in one of the rooms and then take them to the escape pod room where they then use them and win the game. I made a method in the Game class which tests this win condition every time an item is used. The method checks that the player is in the escape pod room and that they are trying to use the codes. If both of these conditions are true, then a win condition variable is set to true. Then, the value of this variable is checked in the main while loop for the game. If the win condition is true, the loop exits, a congratulations string is printed, and the game exits.

**Implement a command “back” that takes you back to the last room you’ve been in.**

I added a field to the Room class which tracks the previous room. Before the player moves into a new room, the previous room field is to current room using a mutator method in the Room class, then the current room is updated to the next room. Then, when the back command is received, a method in the Game class runs which checks that the previous room isn’t null, and then moves the player into the previous room.

**Add at least four new commands (in addition to those that were present in the code you got from us).**

The new commands I implemented were: pickup, use, inventory, drop, back, timer, give. To add these commands, I extended the array of commands in the CommandWord class. Then I had to extend the processCommand method in the Game class to execute these new commands. The pickup command allows the user to take items from the room and them to their backpack. The use command adds time to game timer if there is a bonus time for the particular item. The inventory command prints out a string of all of the items in the player’s backpack to the console. Drop allows the user to return items to room and remove them from their backpack. I explained the functionality of the back command above, but it moves the player back into the room they had previously been in. The timer command prints out a string of the time remaining before the oxygen supply in the space station runs out. The give command is one of the challenge tasks and allows the player to give an item in their backpack to one of the other characters in the game.

**Challenge Tasks:**

**Add characters to your game. Characters are also in rooms. Characters can move around by themselves.**

I created a new Character class. The class is similar to the player class because it has a name field, and a backpack. A currentRoom field holds the room that specific character is in. An array list of possible names is used so the name of the characters could be randomised. There are accessor and mutator methods for the name, and currentRoom fields. Adding backpack items is also possible to allow the functionality of the “give” command. A method in Game makes a random number of characters up to a maximum and another method prints out which characters are in the room with the player. To get the characters to move, I made a method called moveCharacter in Game, which randomly one of the available exits in the room the character is in and then sets the characters currentRoom field to the room through that exit. This is called every time the player moves so that all the characters can move about through the game.

**Extend the parser to recognise three-word commands.**

The “give” command takes 3 words. This command allows the user to take an item in their backpack and give it to a character in the same room as them. The item is removed from the player’s backpack and added to the character’s backpack. To work with three-word commands, I extended the Parser class so for each command if there wasn’t a third word, it returned null, but if there was, it returned the word. Then in the give item method in Game, I check that there are 3 words in the command. If the player has the item, and they are in the same room as the character they want to give the item to, the item is exchanged. Otherwise, it fails and a string explaining why it didn’t work is printed to the user.

**Add a magic transporter room.**

I created a new room called the transporter room. Then, in the method that handles the user moving between rooms, I included a conditional statement that checks if the next room is the transporter room. If it is, a string explaining to the user that they went into the transporter room is printed and a randomly selected room out of an ArrayList of Room objects is selected to be the new current room for the player. The back command still works with this added functionality as it returns them to room they were in before they went into the transporter room.

**Others. You can invent additional challenge tasks yourself.**

I also added a timer to the game. This was done in a separate class called timer. I added a timer because it adds more of a challenge to the game. The timer is initialised to only be 3 minutes but with the use of items, the player can make the game last longer while they try and find the codes. As I had not implemented a timer before, I used some code from Stack Overflow, which I have referenced and credited in the comment of the methods. I also extended the code from the forum further so that it worked for my game. I added methods to print out the time as a string and add bonus time.

**Code Quality Considerations:**

**Coupling:**

I considered coupling when I moved some of the functionality of the Game class into a new class. I created a new class, called CommandExecuter that handled the execution of commands. However, for this to work, the Command Executer class had to be linked to the Game, Room, Player and Items classes. Removing the methods from the Game class didn’t remove the links that it had to those classes either, so there were multiple connections between all of these classes. As a result, changes in the implementation of one of the classes had a ripple effect on the other classes. So, I decided to keep the execution of commands as part of the Game class and reduce the level of coupling.

**Cohesion:**

I considered the level of cohesion in my project when I wanted to add a timer. At first, I added the timer as methods in the Game class because that was the only class were the data from a timer would be needed. However, the timer is a separate entity to the Game class and it would mean the Game class would be handling the main loop of the game and executing commands in the game as well as the timer, so I decided to make the timer into a separate class.

**Responsibility-driven design:**

I considered this when I was adding the “pickup” command. Initially, I created an Items object, and checked that the weight of the item was less than the space in the backpack by storing the space in the backpack in a variable, and then calling a method of the Player class to add the item. But I thought that the Game class shouldn’t be working this out, when the Player class has all the relevant data and I can pass the item weight as a parameter. So, I changed the implementation of the pickup command so that the available space is computed in the Player class rather than in the Game class.

**Maintainability:**

To ensure that the code is maintainable, I documented my code thoroughly. There are documentation comments for every method and class which states the purpose of the method or class as well as any parameters or return types. Also, if I did anything that is unclear or complicated, I tried to document that in the methods with comments.

**Walkthrough:**

The game has a number of random features so there is not one set of commands that will always lead to the player winning but I will explain how the player is most likely to win.

Main Hall

Escape Pods

Transporter

Kitchen

Medical Bay

Reactor

Gym

Store Room

Laboratory

Living Quarters

Room Map:

Above is a room map that details where each room is. The player starts in the living quarters and must get to the escape pods with the codes before the 3 minutes is up. To get to the escape pods the player could enter: *go south, go west, go west,* for example. On the way, the player may find the codes in the room and can pick them up with this command: *pickup codes*. If the player can’t find the codes on the way to the escape pods, they will have to explore around to try and find them. If they are running low on time, they can pick up oxygen tanks, water or a banana and use them (e.g. *use banana*) to add some bonus time to the timer. Once the player has found the codes and brought them to the escape pods, the player wins by using them: *use codes*.

**Known bugs or problems:**

From my own testing, I have fixed all of bugs and problems that I have been able to find.

**Source code for all classes:**

**Game:**

import java.util.ArrayList;

import java.util.Random;

import java.util.HashMap;

/\*\*

\* This class is the main class of the "Zuul Station" application.

\* "Zuul Station" is a simple, text based adventure game. Users

\* can walk around a space station.

\*

\* To play this game, create an instance of this class and call the "play"

\* method.

\*

\* This main class creates and initialises all the others: it creates all

\* rooms, creates the parser and starts the game. It also evaluates and

\* executes the commands that the parser returns.

\*

\* @author Michael Kölling and David J. Barnes

\* @version 2016.02.29

\*

\* Edited and extended by Harry Tennent 25/11/18

\*

\*/

public class Game

{

private Parser parser;

private Room currentRoom;

private long startTime;

private Player player;

private Timer timer;

private boolean winCondition;

private ArrayList<Room> roomsList;

private HashMap<String, Character> charactersMap;

private Character character;

private Random randomGenerator;

/\*\*

\* Create the game and initialise its internal map.

\*/

public Game()

{

randomGenerator = new Random();

roomsList = new ArrayList<>(createRooms());

charactersMap = new HashMap<>();

parser = new Parser();

player = new Player();

timer = new Timer();

winCondition = false;

createRandomCharacters(21); //up to 20 characters randomly in game

}

/\*\*

\* Create all the rooms and link their exits together.

\* @return An array list of all of the room objects.

\*/

private ArrayList<Room> createRooms()

{

Room livingQuarters, medicalBay, mainHall,

laboratory, reactor, escapePodRoom, transporterRoom,

kitchen, storeRoom, gym;

ArrayList<Room> rooms = new ArrayList<>();

// create the rooms

livingQuarters = new Room("in the living quarters");

medicalBay = new Room("in the station's medical bay");

mainHall = new Room("in the main hall");

laboratory = new Room("in the laboratory");

reactor = new Room("in the engine and reactor room");

escapePodRoom = new Room("in the escape pod room");

transporterRoom = new Room("in the transporter room");

kitchen = new Room("in the kitchen");

storeRoom = new Room("in the store room");

gym = new Room("in the gym");

// initialise room exits

livingQuarters.setExit("west", medicalBay);

livingQuarters.setExit("south", laboratory);

livingQuarters.setExit("north", mainHall);

medicalBay.setExit("east", livingQuarters);

medicalBay.setExit("west", transporterRoom);

medicalBay.setExit("north", kitchen);

mainHall.setExit("south", livingQuarters);

mainHall.setExit("west", kitchen);

laboratory.setExit("north", livingQuarters);

laboratory.setExit("west", reactor);

laboratory.setExit("south", storeRoom);

reactor.setExit("east", laboratory);

reactor.setExit("west", escapePodRoom);

reactor.setExit("south", gym);

reactor.setExit("north", medicalBay);

escapePodRoom.setExit("east", reactor);

transporterRoom.setExit("east", medicalBay);

kitchen.setExit("east", mainHall);

kitchen.setExit("south", medicalBay);

storeRoom.setExit("north", laboratory);

storeRoom.setExit("west", gym);

gym.setExit("north", reactor);

gym.setExit("east", storeRoom);

//add rooms to room list

rooms.add(livingQuarters);

rooms.add(medicalBay);

rooms.add(mainHall);

rooms.add(laboratory);

rooms.add(reactor);

rooms.add(escapePodRoom);

rooms.add(transporterRoom);

rooms.add(kitchen);

rooms.add(storeRoom);

rooms.add(gym);

//stop more than 1 set of codes being created in the game

ArrayList<Room> roomsWithCodes = new ArrayList<>();

int roomWithCodesCounter = 0;

for (int i = 0; i < rooms.size(); i++) {

boolean hasCodes = rooms.get(i).hasCodes();

if (hasCodes) {

roomsWithCodes.add(rooms.get(i));

roomWithCodesCounter++;

}

if (roomWithCodesCounter > 1) {

rooms.get(i).removeItemFromRoom("codes");

roomWithCodesCounter--;

}

}

currentRoom = livingQuarters; // start game in the livingQuarters

return rooms;

}

/\*\*

\* Main play routine. Loops until end of play.

\*/

public void play()

{

printWelcome();

// Enter the main command loop. Here we repeatedly read commands and

// execute them until the game is over.

timer.startTimer();

boolean finished = false;

while ((!finished) && (timer.getTimeRemaining() > 0) && (!winCondition)) {

Command command = parser.getCommand();

finished = processCommand(command);

}

if (winCondition) {

System.out.println("Congratulations! You found the codes and" +

" reached the escape pods before time ran out.");

timer.printTimeRemaining();

} else if (timer.getTimeRemaining() <= 0) {

System.out.println("You ran out of time and your " +

"oxygen supply is empty! Better luck next time.");

}

System.out.println("Thank you for playing. Good bye.");

}

/\*\*

\* Print out the opening message for the player.

\*/

private void printWelcome()

{

System.out.println();

System.out.println("Welcome to the World of Zuul!");

System.out.println("World of Zuul is a new text-base adventure game.");

System.out.println("This is set in a space station, " +

"hundreds of thousands of miles away from Earth.\nThe space " +

"station has been in a crash and the oxygen supply to the "+

"station has \nbeen cut off. To win, you must find the codes to release " +

"the escape pods.\n" + "The existing oxygen supply runs out in 3 minutes.\n" +

"Starting now.");

System.out.println("Type 'help' if you need help.");

System.out.println();

System.out.println(currentRoom.getLongDescription(""));

}

/\*\*

\* Given a command, process (that is: execute) the command.

\* @param command The command to be processed.

\* @return true If the command ends the game, false otherwise.

\*/

private boolean processCommand(Command command)

{

boolean wantToQuit = false;

if(command.isUnknown()) {

System.out.println("I don't know what you mean...");

return false;

}

String commandWord = command.getCommandWord();

if (commandWord.equals("help")) {

printHelp();

} else if (commandWord.equals("go")) {

currentRoom = goRoom(command);

printCharactersInCurrentRoom();

} else if (commandWord.equals("quit")) {

wantToQuit = quit(command);

} else if (commandWord.equals("pickup")) {

pickupItem(command);

} else if (commandWord.equals("use")) {

if (testWinCondition(command)) {

return testWinCondition(command);

} else {

timer.addBonusTime(useItem(command)); //useItem returns extra time from using item

}

} else if (commandWord.equals("inventory")) {

showInventory();

} else if (commandWord.equals("drop")) {

dropItem(command);

} else if (commandWord.equals("back")) {

back();

} else if (commandWord.equals("timer")) {

timer.printTimeRemaining();

} else if (commandWord.equals("give")) {

giveItem(command);

}

// else command not recognised.

return wantToQuit;

}

/\*\*

\* Given a max. number of characters, create a random number

\* of characters and put them in the game.

\* @param numberOfCharacters Maximum number of characters

\*/

private void createRandomCharacters(int numberOfCharacters) {

//random number + 1 so there is always at least 1 character in the game

int randNumber = randomGenerator.nextInt(numberOfCharacters) + 1;

int i = 0;

while (i < randNumber) {

character = new Character();

charactersMap.put(character.getName(), character);

//add character to a random room

int randIndex = randomGenerator.nextInt(roomsList.size());

character.setCurrentRoom(roomsList.get(randIndex));

i++;

}

}

/\*\*

\* Creates and prints a string containin the names of all of the

\* characters in the same room as the player.

\*/

private void printCharactersInCurrentRoom() {

String charactersInRoom = "";

for (Character character : charactersMap.values()) {

if (character.getCurrentRoom().equals(currentRoom)) {

//player is in the same room as the character

charactersInRoom += character.getName() + " ";

} //other print out an empty string

}

System.out.println(currentRoom.getLongDescription(charactersInRoom));

}

//implementation of user commands

/\*\*

\* Print out some help information.

\* Here we print some details of the aim of the game and a list of the

\* command words.

\*/

private void printHelp()

{

System.out.println("You have to try and find the escape pod codes.");

System.out.println("Then, you must take them to the escape pod room");

System.out.println("and use them before the oxygen supply runs out.");

System.out.println("Good luck!");

System.out.println();

System.out.println("Your command words are:");

parser.showCommands();

}

/\*\*

\* Checks whether the player has the codes in their backpack, and checks

\* that they are in the escape pod room. If both of these conditions are true,

\* then the win condition is set to true, otherwise, it is false.

\* @param command The command to be processed.

\* @return true If the player is in the escape pod room with the codes, false otherwise.

\*/

private boolean testWinCondition(Command command) {

String item = command.getSecondWord();

boolean inEscapeRoom = currentRoom.getShortDescription().equals("in the escape pod room");

boolean hasCodes = false;

if (player.getBackpackItem(item) != null) {

hasCodes = player.getBackpackItem(item).equals("codes");

}

if (player.getBackpackItem(item) != null && hasCodes && inEscapeRoom) {

winCondition = true;

return winCondition;

}

return false; //only get to this point if the condition was not met

}

/\*\*

\* Try to go in to one direction. If there is an exit, enter the new

\* room, otherwise print an error message. If the player enters, the

\* transporter room, randomly pick a room to move them to.

\* @param command Command stating which direction to go in.

\* @return A room object of the room the player is currently in.

\*/

private Room goRoom(Command command)

{

if(!command.hasSecondWord()) {

// if there is no second word, we don't know where to go...

System.out.println("Go where?");

return currentRoom;

}

String direction = command.getSecondWord();

// Try to leave current room.

Room nextRoom = currentRoom.getExit(direction);

if (nextRoom == null) {

System.out.println("There is no door!");

} else if (nextRoom.getShortDescription().equals("in the transporter room")) {

System.out.println("You entered the broken transporter room!");

int randInt = randomGenerator.nextInt(roomsList.size());

Room randomRoom = roomsList.get(randInt);

randomRoom.setPreviousRoom(currentRoom);

currentRoom = randomRoom;

}

else {

nextRoom.setPreviousRoom(currentRoom);

currentRoom = nextRoom; //player moves to another room

}

return currentRoom;

}

/\*\*

\* This method is called every time the player calls the 'go' method,

\* and it moves the characters in the game in a random direction.

\* @param character Character object for the character to be moved.

\*/

private void moveCharacter(Character character) {

ArrayList<String> exitList = character.getCurrentRoom().getExits();

int randIndex = randomGenerator.nextInt(exitList.size()); //north,east,south,west

String randomDirection = exitList.get(randIndex);

Room nextRoom = character.getCurrentRoom().getExit(randomDirection);

if (nextRoom != null) {

character.setCurrentRoom(nextRoom);

}

}

/\*\*

\* A method to pick an item up from the room the player is in

\* and add it to their backpack, provided that they have enough

\* space and the item can be picked up.

\* @param command Command containing the item to pick up.

\*/

private void pickupItem(Command command) {

if(!command.hasSecondWord()) {

// if there is no second word, we don't know what to pick up...

System.out.println("Pick up what?");

return;

}

String item = command.getSecondWord();

Items roomItem = new Items();

if (roomItem.isAnItem(item)) {

int itemWeight = roomItem.getWeight(item);

if (item.equals(null)) {

System.out.println("There is no " + item + " to pick up.");

} else if (roomItem.getCanBePickedUp(item)) {

player.addBackpackItem(item, itemWeight);

currentRoom.removeItemFromRoom(item);

} else {

System.out.println("You can't pick this item up.");

}

} else {

System.out.println(item + " is not an item.");

}

}

/\*\*

\* If the specified item has bonus time, add this to the game clock and

\* remove it from the player's backpack.

\* @param command Command containing the item to be used.

\* @return An int of the bonus time.

\*/

private int useItem(Command command) {

if(!command.hasSecondWord()) {

// if there is no second word, we don't know what to pick up...

System.out.println("Use what?");

return 0;

}

String item = command.getSecondWord();

Items roomItem = new Items();

if (player.getBackpackItem(item) != null) {

int bonusTime = roomItem.getBonusTime(item);

int itemWeight = roomItem.getWeight(item);

if (bonusTime == 0) {

System.out.println("There is no bonus time for using this item.");

return 0;

}

player.removeBackpackItem(item, itemWeight);

return bonusTime;

} else {

System.out.println("You don't have that item in your backpack.");

return 0;

}

}

/\*\*

\* Print out the contents of the player's backpack.

\*/

private void showInventory() {

player.printBackpackItems();

}

/\*\*

\* Remove item from the player's backpack and return it to the room.

\* @param command Command containing the item to drop.

\*/

private void dropItem(Command command) {

if(!command.hasSecondWord()) {

// if there is no second word, we don't know what to drop...

System.out.println("Drop what?");

return;

}

String item = command.getSecondWord();

Items roomItem = new Items();

if (player.getBackpackItem(item) != null) {

int itemWeight = roomItem.getWeight(item);

player.removeBackpackItem(item, itemWeight);

System.out.println(item + " removed from your backpack.");

currentRoom.returnItemToRoom(item);

} else {

System.out.println("You don't have that item in your backpack.");

return;

}

}

/\*\*

\* Take the player back to the room they were previously in.

\*/

private void back() {

// Try to leave current room.

Room nextRoom = currentRoom.getPreviousRoom();

if (nextRoom == null) {

System.out.println("You haven't been anywhere yet!");

}

else {

currentRoom.setPreviousRoom(currentRoom);

currentRoom = nextRoom;

System.out.println(currentRoom.getLongDescription(""));

}

}

/\*\*

\* A three-word command that allows the player to give items in their

\* backpack to other characters in the game, provided they are in the same

\* room.

\* @param command Command contain the item to give and the person to give it to.

\*/

private void giveItem(Command command) {

if(!command.hasSecondWord()) {

// if there is no second word, we don't know what to give...

System.out.println("Give what?");

return;

} else if (!command.hasThirdWord()) {

System.out.println("Give to who?");

return;

}

String item = command.getSecondWord();

String name = command.getThirdWord();

Character character = charactersMap.get(name);

Items roomItem = new Items();

int itemWeight;

//checks player has that item and is in the same room as the character

if ((player.getBackpackItem(item) != null) &&

(character.getCurrentRoomString().equals(currentRoom.getShortDescription()))) {

character.addBackpackItem(item);

itemWeight = roomItem.getWeight(item);

player.removeBackpackItem(item, itemWeight);

System.out.println(item + " given to " + character.getName());

} else if (!character.getCurrentRoomString().equals(currentRoom.getShortDescription())) {

System.out.println(character.getName() + " is not in this room.");

} else if (player.getBackpackItem(item) == null){

System.out.println("You do not have this item in your backpack.");

} else {

System.out.println("You cannot give this item.");

}

}

/\*\*

\* "Quit" was entered. Check the rest of the command to see

\* whether we really quit the game.

\* @return true, if this command quits the game, false otherwise.

\*/

private boolean quit(Command command)

{

if(command.hasSecondWord()) {

System.out.println("Quit what?");

return false;

}

else {

return true; // signal that we want to quit

}

}

}

**Room:**

import java.util.Set;

import java.util.HashMap;

import java.util.Random;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* Class Room - a room in an adventure game.

\*

\* This class is part of the "Zuul Station" application.

\* "Zuul Station" is a simple, text based adventure game.

\*

\* A "Room" represents one location in the scenery of the game. It is

\* connected to other rooms via exits. For each existing exit, the room

\* stores a reference to the neighboring room. There are also items in

\* room which can be picked up and dropped, adding and removing them

\* from the room respectively.

\*

\* @author Michael Kölling and David J. Barnes

\* @version 2016.02.29

\*

\* Extended and edited by Harry Tennent 25/11/18.

\*/

public class Room

{

private String description;

private Room previousRoom;

private HashMap<String, Room> exits; // stores exits of this room.

private HashMap<String, Items> itemsInRoom;

private Random randomGenerator;

private Items roomItem;

private boolean hasCodes;

/\*\*

\* Create a room described "description". Initially, it has

\* no exits. "description" is something like "a kitchen" or

\* "an open court yard".

\* @param description The room's description.

\*/

public Room(String description)

{

this.description = description;

previousRoom = null;

hasCodes = false;

exits = new HashMap<>();

randomGenerator = new Random();

itemsInRoom = new HashMap<>();

createRoomItems(6);

}

/\*\*

\* Given a max number of items for the room, create a random number

\* of items and add them to the room.

\* @param itemNumber Max number of items.

\*/

private void createRoomItems(int itemNumber) {

int randItemNumber = randomGenerator.nextInt(itemNumber); //up to 5 items in room

int i = 0;

while (i <= randItemNumber) {

roomItem = new Items();

roomItem.getRandomItem();

if (roomItem.getName().equals("codes")) {

hasCodes = true;

}

itemsInRoom.put(roomItem.getName(), roomItem);

i++;

}

}

/\*\*

\* Remove item from room.

\* @param item Item to remove.

\*/

public void removeItemFromRoom(String item) {

itemsInRoom.remove(item);

}

/\*\*

\* Put item back into the room.

\* @param itemName Name of item to add back.

\*/

public void returnItemToRoom(String itemName) {

roomItem = new Items();

roomItem.getItem(itemName);

itemsInRoom.put(roomItem.getName(), roomItem);

}

/\*\*

\* Define an exit from this room.

\* @param direction The direction of the exit.

\* @param neighbor The room to which the exit leads.

\*/

public void setExit(String direction, Room neighbor)

{

exits.put(direction, neighbor);

}

/\*\*

\* @return The short description of the room

\* (the one that was defined in the constructor).

\*/

public String getShortDescription()

{

return description;

}

/\*\*

\* Return a description of the room in the form:

\* You are in the kitchen.

\* Exits: north west

\* Items in the room: banana

\* Characters in the room: PrivateRyan

\* @return A long description of this room

\*/

public String getLongDescription(String charactersInRoom)

{

String roomItemsString = "\nItems in the room: ";

String characterString = "\nCharacters in the room: ";

String longDescription = "You are " + description +

".\n" + getExitString();

for (String item : itemsInRoom.keySet()) {

roomItemsString += (item + " ");

}

characterString += charactersInRoom;

longDescription += roomItemsString + characterString;

return longDescription;

}

/\*\*

\* Return a string describing the room's exits, for example

\* "Exits: north west".

\* @return Details of the room's exits.

\*/

private String getExitString()

{

String returnString = "Exits:";

Set<String> keys = exits.keySet();

for(String exit : keys) {

returnString += " " + exit;

}

return returnString;

}

/\*\*

\* Return the room that is reached if we go from this room in direction

\* "direction". If there is no room in that direction, return null.

\* @param direction The exit's direction.

\* @return The room in the given direction.

\*/

public Room getExit(String direction)

{

return exits.get(direction);

}

/\*\*

\* Return an array list of all the exits in the room.

\* @return Array list of the room's exits.

\*/

public ArrayList<String> getExits() {

ArrayList<String> exitList = new ArrayList<>();

Set<String> keys = exits.keySet();

for(String exit : keys) {

exitList.add(exit);

}

return exitList;

}

/\*\*

\* Store the previous room the player was in.

\* @param room Previous room object.

\*/

public void setPreviousRoom(Room room) {

previousRoom = room;

}

/\*\*

\* Return the previous room the player was in.

\* @return Previous room object.

\*/

public Room getPreviousRoom() {

return previousRoom;

}

public boolean hasCodes() {

return hasCodes;

}

}

**Player:**

import java.util.ArrayList;

/\*\*

\* Represents the player in the game. Keeps track of the items

\* in their backpack, total weight of the backpack and provides

\* methods to add, remove and print items of the backpack.

\*

\* @author Harry Tennent

\* @version 15/11/18

\*/

public class Player

{

// instance variables

private static final int MAX\_WEIGHT = 100;

private ArrayList<String> backpack;

private int currentBackpackWeight;

/\*\*

\* Constructor for objects of class Player

\*/

public Player()

{

// initialise instance variables

backpack = new ArrayList<>();

currentBackpackWeight = 0;

}

/\*\*

\* Returns the backpack item at the specified index.

\* @param index Index of item to return.

\* @return String Backpack item.

\*/

public String getBackpackItem(int index) {

return backpack.get(index);

}

/\*\*

\* Returns the backpack item matching the given item name.

\* @param itemName Name of the item to return.

\* @return String Backpck item if found or null otherwise.

\*/

public String getBackpackItem(String itemName) {

if (backpack.contains(itemName)) {

return backpack.get(backpack.indexOf(itemName));

} else {

return null;

}

}

/\*\*

\* Add an item to the player's backpack.

\* @param itemName Name of the item to add.

\* @param itemWeight Weight of the item to add.

\*/

public void addBackpackItem(String itemName, int itemWeight) {

int availableSpace = MAX\_WEIGHT - currentBackpackWeight;

if (availableSpace >= itemWeight) {

backpack.add(itemName);

currentBackpackWeight += itemWeight;

System.out.println(itemName + " added to your backpack!");

} else {

System.out.println("You don't have enough space!");

}

}

/\*\*

\* Remove an item from the player's backpack.

\* @param itemName Name of the item to add.

\* @param itemWeight Weight of the item to add.

\*/

public void removeBackpackItem(String itemName, int itemWeight) {

currentBackpackWeight -= itemWeight;

if (backpack.contains(itemName)) {

backpack.remove(itemName);

} else {

System.out.println("That item is not in your backpack.");

}

}

/\*\*

\* Returns the space left in the backpack.

\* @return int of available space.

\*/

public int spaceAvailable() {

return MAX\_WEIGHT - currentBackpackWeight;

}

/\*\*

\* Print a string of all the items in the backpack to the console.

\*/

public void printBackpackItems() {

String inventoryString = "Items in backpack: ";

for (String i : backpack) {

inventoryString += (i + " ");

}

System.out.println(inventoryString);

}

}

**Items:**

import java.util.HashMap;

import java.util.ArrayList;

import java.util.Random;

import java.util.Set;

/\*\*

\* Items represent all of the possible items in the game.

\* Specific items or a random item can be constructed.

\* Every item has a field for name, weight, whether it can

\* be picked up, and any bonus time assosciated with having

\* used it.

\* This information is stored in a HashMap of a String key and

\* a List of Strings as its value.

\*

\* @author Harry Tennent

\* @version 25/11/18

\*/

public class Items

{

private Random randomGenerator;

private String itemName;

private int itemWeight;

private boolean canBePickedUp;

private int bonusTime; //added time when item is used

private HashMap<String, ArrayList<String>> itemHashMap;

/\*\*

\* Constructor for objects of class Items. Adds the item

\* data to the hash map.

\*/

public Items()

{

randomGenerator = new Random();

itemHashMap = new HashMap<String, ArrayList<String>>();

ArrayList<String> oxygenTankData = new ArrayList<String>();

oxygenTankData.add("40"); //weight

oxygenTankData.add("true"); //can be picked up

oxygenTankData.add("30"); //bonus time

ArrayList<String> codesData = new ArrayList<String>();

codesData.add("50"); //weight

codesData.add("true"); //can be picked up

codesData.add("0"); //bonus time

ArrayList<String> bananaData = new ArrayList<String>();

bananaData.add("10"); //weight

bananaData.add("true"); //can be picked up

bananaData.add("15"); //bonus time

ArrayList<String> waterData = new ArrayList<String>();

waterData.add("20"); //weight

waterData.add("true"); //can be picked up

waterData.add("15"); //bonus time

ArrayList<String> chairData = new ArrayList<String>();

chairData.add("20"); //weight

chairData.add("false"); //can be picked up

chairData.add("0"); //bonus time

ArrayList<String> plantData = new ArrayList<String>();

plantData.add("10"); //weight

plantData.add("false"); //can be picked up

plantData.add("0"); //bonus time

ArrayList<String> tableData = new ArrayList<String>();

tableData.add("30"); //weight

tableData.add("false"); //can be picked up

tableData.add("0"); //bonus time

ArrayList<String> bookData = new ArrayList<String>();

bookData.add("10"); //weight

bookData.add("true"); //can be picked up

bookData.add("0"); //bonus time

ArrayList<String> lampData = new ArrayList<String>();

lampData.add("20"); //weight

lampData.add("false"); //can be picked up

lampData.add("0"); //bonus time

ArrayList<String> tabletData = new ArrayList<String>();

tabletData.add("15"); //weight

tabletData.add("false"); //can be picked up

tabletData.add("0"); //bonus time

ArrayList<String> cupboardData = new ArrayList<String>();

cupboardData.add("40"); //weight

cupboardData.add("false"); //can be picked up

cupboardData.add("0"); //bonus time

itemHashMap.put("oxygenTank", oxygenTankData);

itemHashMap.put("codes", codesData);

itemHashMap.put("banana", bananaData);

itemHashMap.put("water", waterData);

itemHashMap.put("chair", chairData);

itemHashMap.put("plant", plantData);

itemHashMap.put("table", tableData);

itemHashMap.put("book", bookData);

itemHashMap.put("lamp", lampData);

itemHashMap.put("tablet", tabletData);

itemHashMap.put("cupboard", cupboardData);

}

/\*\*

\* Randomly assign's an items properties to the

\* Items fields.

\*/

public void setRandomItemProperties() {

Set<String> keySet = itemHashMap.keySet();

ArrayList<String> keyList = new ArrayList<>();

for (String i : keySet) {

keyList.add(i);

}

int randIndex = randomGenerator.nextInt(keyList.size());

itemName = keyList.get(randIndex);

itemWeight = getWeight(itemName);

canBePickedUp = getCanBePickedUp(itemName);

bonusTime = getBonusTime(itemName);

}

/\*\*

\* Calls setRandomItemProperties().

\*/

public void getRandomItem() {

setRandomItemProperties();

}

/\*\*

\* Sets the Items object's field to the data

\* assosciated with the item name given.

\* @param itemName Name of the item.

\*/

public void getItem(String itemName) {

this.itemName = itemName;

itemWeight = getWeight(itemName);

canBePickedUp = getCanBePickedUp(itemName);

bonusTime = getBonusTime(itemName);

}

/\*\*

\* Returns item name.

\* @return Name of the item if it is an item, null otherwise.

\*/

public String getName() {

if (itemHashMap.containsKey(itemName)) {

return itemName;

} else {

return null;

}

}

/\*\*

\* Returns the weight of the item.

\* @param Name of the item to get the weight for.

\* @return Item weight.

\*/

public int getWeight(String itemName) {

if (itemHashMap.containsKey(itemName)) {

String weight = itemHashMap.get(itemName).get(0); //Value 0 is item weight

return Integer.parseInt(weight);

} else {

return 0;

}

}

/\*\*

\* Returns whether the item can be picked up or not.

\* @param Name of the item.

\* @return true if the item can be picked up, false otherwise.

\*/

public boolean getCanBePickedUp(String itemName) {

if (itemHashMap.containsKey(itemName)) {

return Boolean.parseBoolean(itemHashMap.get(itemName).get(1));//Value 1 is pick up boolean.

} else {

return false;

}

}

/\*\*

\* Returns the bonus time of the item, if any.

\* @param itemName Name of the item.

\* @return Bonus time, if any.

\*/

public int getBonusTime(String itemName) {

if (itemHashMap.containsKey(itemName)) {

return Integer.parseInt(itemHashMap.get(itemName).get(2)); //Value 2 is bonus time.

} else {

return 0;

}

}

/\*\*

\* A check to see if the specified item is an actual item.

\* @param itemName Name of the item.

\* @return true if the item is an item, false otherwise.

\*/

public boolean isAnItem(String itemName) {

if (itemHashMap.containsKey(itemName)) {

return true;

} else {

return false;

}

}

}

**Character:**

import java.util.ArrayList;

import java.util.Random;

/\*\*

\* Character class represents other characters in the game.

\* They can walk throughout the game. The player can give them items

\* if they wish to.

\*

\* @author Harry Tennent

\* @version 25/11/18

\*/

public class Character

{

// instance variables

ArrayList<String> names;

Random randomGenerator;

String name;

Room currentRoom;

ArrayList<String> backpack;

/\*\*

\* Constructor for objects of class Character

\*/

public Character()

{

// initialise instance variables

randomGenerator = new Random();

names = new ArrayList<>();

names.add("CommanderBob");

names.add("PrivateDylan");

names.add("PrivateRyan");

names.add("LieutenantFiona");

names.add("CaptainJane");

currentRoom = null;

setName();

backpack = new ArrayList<>();

}

/\*\*

\* Return name of the character.

\* @return Character name.

\*/

public String getName()

{

return name;

}

/\*\*

\* Random pick a name for the character.

\*/

private void setName() {

int randIndex = randomGenerator.nextInt(names.size());

name = names.get(randIndex);

}

/\*\*

\* Set the current room the character is in.

\* @param room Room object the character is in.

\*/

public void setCurrentRoom(Room room) {

currentRoom = room;

}

/\*\*

\* Return the room the character is in.

\* @return Current room object.

\*/

public Room getCurrentRoom() {

return currentRoom;

}

/\*\*

\* Returns a string of the current room's description.

\* @return Current room's description.

\*/

public String getCurrentRoomString() {

return currentRoom.getShortDescription();

}

/\*\*

\* Add an item to the backpack.

\* @param itemName Name of item to add.

\*/

public void addBackpackItem(String itemName) {

backpack.add(itemName);

}

}

**CommandWords:**

/\*\*

\* This class is part of the "Zuul Station" application.

\* "Zuul Station" is a simple, text based adventure game.

\*

\* This class holds an enumeration of all command words known to the game.

\* It is used to recognise commands as they are typed in.

\*

\* @author Michael Kölling and David J. Barnes

\* @version 2016.02.29

\*

\* Extended and edited by Harry Tennent 25/11/18.

\*/

public class CommandWords

{

// a constant array that holds all valid command words

private static final String[] validCommands = {

"go", "quit", "help", "pickup", "use", "inventory", "drop",

"back", "timer", "give"

};

/\*\*

\* Constructor - initialise the command words.

\*/

public CommandWords()

{

// nothing to do at the moment...

}

/\*\*

\* Check whether a given String is a valid command word.

\* @return true if it is, false if it isn't.

\*/

public boolean isCommand(String aString)

{

for(int i = 0; i < validCommands.length; i++) {

if(validCommands[i].equals(aString))

return true;

}

// if we get here, the string was not found in the commands

return false;

}

/\*\*

\* Print all valid commands to System.out.

\*/

public void showAll()

{

for(String command: validCommands) {

System.out.print(command + " ");

}

System.out.println();

}

}

**Command:**

/\*\*

\* This class is part of the "Zuul Station" application.

\* "Zuul Station" is a simple, text based adventure game.

\*

\* This class holds information about a command that was issued by the user.

\* A command currently consists of three strings: a command word, a second

\* word (for example, if the command was "take map", then the two strings

\* obviously are "take" and "map"), and a third word (e.g. give water PrivateRyan).

\*

\* The way this is used is: Commands are already checked for being valid

\* command words. If the user entered an invalid command (a word that is not

\* known) then the command word is <null>.

\*

\* If the command had only one word, then the second and third words are <null>.

\*

\* @author Michael Kölling and David J. Barnes

\* @version 2016.02.29

\*

\* Extended and edited by Harry Tennent 25/11/18.

\*/

public class Command

{

private String commandWord;

private String secondWord;

private String thirdWord;

/\*\*

\* Create a command object. First, second, third word must be supplied, but

\* any (or all) can be null.

\* @param firstWord The first word of the command. Null if the command

\* was not recognised.

\* @param secondWord The second word of the command.

\* @param thirdWord The third word of the command.

\*/

public Command(String firstWord, String secondWord, String thirdWord)

{

commandWord = firstWord;

this.secondWord = secondWord;

this.thirdWord = thirdWord;

}

/\*\*

\* Return the command word (the first word) of this command. If the

\* command was not understood, the result is null.

\* @return The command word.

\*/

public String getCommandWord()

{

return commandWord;

}

/\*\*

\* @return The second word of this command. Returns null if there was no

\* second word.

\*/

public String getSecondWord()

{

return secondWord;

}

/\*\*

\* @return The third word of the command. Returns null if there

\* was no third word.

\*/

public String getThirdWord() {

return thirdWord;

}

/\*\*

\* @return true if this command was not understood.

\*/

public boolean isUnknown()

{

return (commandWord == null);

}

/\*\*

\* @return true if the command has a second word.

\*/

public boolean hasSecondWord()

{

return (secondWord != null);

}

/\*\*

\* @return true if the command has a third word.

\*/

public boolean hasThirdWord() {

return (thirdWord != null);

}

}

**Parser:**

import java.util.Scanner;

/\*\*

\* This class is part of the "Zuul Station" application.

\* ""Zuul Station" is a simple, text based adventure game.

\*

\* This parser reads user input and tries to interpret it as an "Adventure"

\* command. Every time it is called it reads a line from the terminal and

\* tries to interpret the line as a three word command. It returns the command

\* as an object of class Command.

\*

\* The parser has a set of known command words. It checks user input against

\* the known commands, and if the input is not one of the known commands, it

\* returns a command object that is marked as an unknown command.

\*

\* @author Michael Kölling and David J. Barnes

\* @version 2016.02.29

\*/

public class Parser

{

private CommandWords commands; // holds all valid command words

private Scanner reader; // source of command input

/\*\*

\* Create a parser to read from the terminal window.

\*/

public Parser()

{

commands = new CommandWords();

reader = new Scanner(System.in);

}

/\*\*

\* @return The next command from the user.

\*/

public Command getCommand()

{

String inputLine; // will hold the full input line

String word1 = null;

String word2 = null;

String word3 = null;

System.out.print("> "); // print prompt

inputLine = reader.nextLine();

// Find up to two words on the line.

Scanner tokenizer = new Scanner(inputLine);

if(tokenizer.hasNext()) {

word1 = tokenizer.next(); // get first word

if(tokenizer.hasNext()) {

word2 = tokenizer.next(); // get second word

if (tokenizer.hasNext()) {

word3 = tokenizer.next(); //get third word

}

// note: we just ignore the rest of the input line.

}

}

// Now check whether this word is known. If so, create a command

// with it. If not, create a "null" command (for unknown command).

if(commands.isCommand(word1)) {

return new Command(word1, word2, word3);

}

else {

return new Command(null, word2, word3);

}

}

/\*\*

\* Print out a list of valid command words.

\*/

public void showCommands()

{

commands.showAll();

}

}

**Timer:**

/\*\*

\* The timer class manages the game timer. It it started as soon

\* as the game starts. Bonus time can be added to the timer by

\* using items. When the timer runs out, the game ends.

\*

\* @author Harry Tennent

\* @version 25/11/18

\*/

public class Timer

{

private long startTime;

private long bonusTime;

private long timeInSeconds;

/\*\*

\* Constructor for objects of class Timer

\*/

public Timer()

{

// initialise instance variables

startTime = 0;

bonusTime = 0;

}

/\*\*

\* Timer code taken from this StackOverflow answer:

\* https://stackoverflow.com/questions/10820033/make-a-simple-timer-in-java/14323134

\* From user "AgilePro" on the 15/11/18.

\*

\* Starts the timer.

\*/

public void startTimer() {

startTime = System.currentTimeMillis();

}

/\*\*

\* Timer code taken and edited from this StackOverflow answer:

\* https://stackoverflow.com/questions/10820033/make-a-simple-timer-in-java/14323134

\* From user "AgilePro" on the 15/11/18.

\*

\* Returns the time remaining on the clock.

\* @return long value of the time remaining in seconds.

\*/

public long getTimeRemaining() {

int timer = 180000; //3 minutes for the game in milliseconds

long timeElapsed = System.currentTimeMillis() - startTime;

long timeRemaining = timer - timeElapsed + bonusTime;

long timeInSeconds = timeRemaining / 1000;

long seconds = timeInSeconds % 60;

long minutes = timeInSeconds / 60;

if (seconds < 0 || minutes < 0) { //so negative numbers don't show

seconds = 0;

minutes = 0;

}

return timeInSeconds;

}

/\*\*

\* Add time to the game timer, print out remaining time.

\* @param time to add to the game timer.

\*/

public void addBonusTime(int time) {

if (time == 0) {

System.out.println("No bonus time was added to the timer.");

printTimeRemaining();

} else {

System.out.println(time + " seconds were added to the timer!");

time = time \* 1000; //convert to milliseconds

bonusTime += time;

printTimeRemaining();

}

}

/\*\*

\* Prints a strings saying how long is left on the game timer.

\*/

public void printTimeRemaining() {

long timeInSeconds = getTimeRemaining();

long seconds = timeInSeconds % 60;

long minutes = timeInSeconds / 60;

System.out.println("Time until the oxygen supply runs out: " + minutes

+ " minutes and " + seconds + " seconds.");

}

}