Programming Practice and Applications, Assignment 2 Report

Name: Christian Impollonia

K number: K1925245

The murder of Melissa Roland

"You are the main detective in a murder case. A mysterious woman named Melissa Roland has been killed in an old mansion. There has been a storm outside for the past 12 hours, so no one could have left the place. The murderer must be one of the people in the mansion. Your objective is to find out who did it, but don't get it wrong! False accusation is a very serious crime here. Good luck."

User level description:

This is a text-based adventure game, in which you can do several things using commands you type. You can move around rooms, study and pick up objects you find and take them with you. You can also interact with characters, by talking to them or by giving them objects you find. The rooms you find are not always accessible, and the items you find are of crucial importance to get through the game. Your objective is to find out what happened inside the mansion, and find the killer by finding concrete evidence.

Implementation description:

The game was implemented by using 9 total classes, with the 'Game' class being the main one. The most significant parts of the game, like characters, items and rooms, all have their own classes. All these objects are created inside the constructor of the main class, when the game is started. Being text-based, the game revolves mainly around the use of HashMaps, that let every one of the created objects be linked to Strings, obtained from player input using the Scanner class. With everything the user does, these objects will be passing around various maps throughout the game, changing what the player can interact with.

Completed base tasks

- The first base task was already implemented in the code. The rooms that were already present were renamed and some were added to suit the theme and story of the game
- The second base task was already implemented in the code. Two other types of exits ("up" and "down") were added, while each room was assigned specific exits based on the game map.
- The third base task was completed by creating a class named Item. Each item has two main characteristics: its weight and if it can be picked up or not. It also has a description, used for a later command. Those are the three fields of the Item class. A method in the Game class was implemented to create the items, similar to the one that creates the rooms. Each room has certain items in it, so a method was also written in the Room class and called in the Game class which adds each item to a HashMap collection, assigned to a Room.
- The fourth base task was implemented by creating a class called Bag. The Bag object has 3 main fields: the current weight, the maximum weight it can hold, and the collection of the items it is holding. Also, the word "pickup" was added to the list of commands in the class CommandWords. In the Game class, the processCommand method was updated to reference a new method, the pickupItem method. The pickupItem method adds items to the HashMap in the bag class and

removes them from the HashMap in the room, only if the item satisfies characteristics like fitting in the bag or being a carriable item. After being added, the weight of the item is added to the weight of the bag.

- Fifth base task: The game can be won by finding out who the killer is. Using the accuse command, the player must select a character in the room and an item in his bag which will be the decisive evidence. The accuseCharacter method checks whether those correspond with a specific character and item, and if they do, the game is won. If not, it is lost. The processCommand method in the game class was in fact modified to add the accuseCharacter method, which will return true both if it is won or lost, stopping the program. To be replayed as intended, the Game object should be removed and created again, considering there is no way to remove that object within the code. If not, the game will restart from where it stopped.
- The sixth base task was implemented by adding a “back” to the command words in the CommandWords class. A method named goBack was created in the Game class, and a call to that method was added to the processCommand method. Also, a field called previousRoom was created, which initially stores the same value as the current room. When the player changes room, the previousRoom field will not change to the new room like the current room does. All the goBack method does is swap the values of the currentRoom and the previousRoom, in practice going to the previous room, and saving the old one as previous. If the command is entered when in the first room, it will simply keep the player in that room and print out the description again.
- Seventh base task: The commands that were added are: pickup, drop, examine, back, talk, use, give, accuse and bag.

  For each of these added commands, a new command word was added in the CommandWords class, a new method was written in the Game class as well as a call for that method in the processCommand method of the same Game class. From now on it is given for granted that each command described below in this bullet point will have followed this same initial process. Accessor and mutator methods were also created in the other classes accordingly.

  Pickup: The pickup command is implemented as explained in the fourth bullet point of base tasks.

  Drop: The drop command was implemented in a very similar way. The dropItem method was created, which does the opposite as the pickUpItem method: it removes an item from the HashMap of items in the bag and adds one to the HashMap of items in the current room. It also decreases the current weight of the bag by the weight of the item that was removed. There is no need for weight checks or canPickUp checks like for the pickup command.

  Examine: The examine command was implemented through the examineItem method. This method retrieves the item from the current room’s hashmap thanks to the keyword inserted by the user, and then uses the printDescription method of that item to print out the detailed description of it. The Item class was modified accordingly for each item to have a description, which is given when the object is created.

  Back: The back command was implemented as described in the previous bullet point.

  Talk: The talk command was implemented in an identical way to the examine command, with the difference that it is characters that print out their line instead of items. The lines characters utter can change if the character is happy or not.

  Use: The use command was implemented by adding two Boolean fields in the Room class, to check if the room is locked or not, or if it is dark or not. Then by modifying the processCommand method in the Game class, so that if a room is locked or dark, most commands are blocked. The player follows the use command with the name of an item from the bag. There are four items that can be used. The key and the flashlight can only be used if the player is in the correct room at the moment of use. The key unlocks rooms, and the flashlight makes them bright. So, the useItem method in the Game class just changes the Boolean fields to false if all the previously mentioned conditions are

met. The stretcher and the beacon have special effects explained in the additional challenge tasks section.
Give: The give command was implemented as explained in the first bullet point of the challenge tasks.
Accuse: The accuse command was implemented as explained in the fifth bullet point of the base tasks.
Bag: The bag command was implemented as explained in the additional challenge tasks section

Completed challenge tasks

- The first challenge task was completed by adding a class called Character, which has three main fields: the name of the character, which is what the player uses to talk to the character, the response, that is the line that they give when talked to, and the current room they are in. It also has two additional fields: the item that makes it happy, and the line it says when happy. In the Room class, a HashMap to store the characters from that room was added, as well as methods to add and remove a character. Characters can move around the map, differently from items. Characters can move only within a specific selection of rooms, because some rooms are locked for story-related reasons.
  A method called moveCharacter was created in the game class, an ArrayList was declared in the method, and obtained from the set of keys in the HashMap of exits for a specific room. It was then adjusted to only have the exits to the accessible rooms. A randomizer was created inside the method, with the bound set as the size of the list.
  Characters choose a random exit to use, and all characters change location every 8 times the player moves from room to room. This number was chosen because, given the number of rooms in the game, it is the most suited to make sure that characters do not feel like they are moving too much. The main method to move the characters, moveCharacter, is called (for all characters) inside the goRoom and goBack methods.
- The second challenge task was completed by updating the Command and Parser class with fields and methods related to a third word. Then a new command was created, the give command. The give word was added to the command words and a method called giveItem was created in the Game class. The giveItem method checks for the presence of both a second and a third word, as well as if the objects exist in that room. Then It proceeds by checking if the item called by the user is the same as the happy item of that character. If it is, it will remove the item from the bag and the character will become happy, with a new response when he's talked to.
- The third challenge task was completed by creating a new room, a magicRoom, and placing an if statement in the goRoom method in the game class. If the next class is the magic room, then by using the Random class, and the array with all available Rooms, one of the 7 rooms is chosen and set as the new current room. The attick room wasn't added to the array list because it is a room that the player can't be allowed to visit before unlocking the bedroom.
- As additional challenge tasks, the possibility to use items was added. Keys and flashlights can be used to unlock door or brighten rooms making them accessible, all done by modifying a Boolean field in the Room class. It is explained in more detail in the seventh bullet point of the base tasks.
  The possibility for objects to be examined and characters to be talked to was also added, with the talk and examine commands explained earlier.
  Another thing that was added was the presence of a stretcher. This item, if picked up and used, will increase the maximum weight the bag can carry by 20. This is done simply by adding that number to the maxWeight field in the bag class, when the use command is used for that item.

Another command was implemented called bag. This will print out the list of items in the player's bag. It is implemented just by printing out the set of keys of the HashMap of items in the bag.
A last additional challenge task completed was the creation of a beacon. The beacon is an item that can be picked up and used to memorize the room the player is in. If it is then used again, the player will be teleported to that room. It was implemented by adding two fields to the Item class: one to memorize the status of the beacon and one to memorize the room, as well as accessors and mutators for them. It is then called from the Game class in the useItem method.

Code quality considerations

Coupling: Loose coupling between classes was considered, to make sure that changing one class in the future will not risk breaking the others. For this reason, all fields in all classes were made private, and accessed by external classes only through accessor methods. (See the Character class for example)
Cohesion: High cohesion was taken into account when making this program. Each class represents a single abstract entity. For example, the Room class represents each room in the game., with specific characteristics in common between every room. Each method is also responsible for one logical task. In the Game class, for example, the processCommand method was not responsible for checking if the room was dark or locked, to see if the command could be executed. A different method called checkRoom was created to do so.
Responsibility-driven design: Each class is responsible for the modification of its own fields. In this game, this was used, for example, in the goRoom method of the Game class. Each time the player moves, the movement count stored in the MainPlayer class increases. This though, is handled by the MainPlayer class with a specific method called increaseMovementCount. All the Game class does is call the method.
Maintainability: To make sure that the game can be maintained, a lot of attention was given to readability and reusability. All fields in fact have meaningful names (see the Game class), and code duplication was avoided when possible. For example in the Game class, looking inside the goRoom and goBack methods, the method moveCharacterEvery was placed. This makes sure that if someone later on wants to change the number of movements needed to move the characters, all they need to do is change the value inside the method, instead of having to do it twice or more times.

Walkthrough

To win the game, follow these steps. Talking and interacting with characters, examining items and the use of special items like the stretcher are optional, but they are the way to find out who the killer is and win the game by yourself. But they are not needed if you use this walkthrough. Type "go west", and then "go north". Then "pickup flashlight" and "back". Now "go east" twice. "Go down" and then "use flashlight". Now that you can see the room, "pickup key" and "back". "Go west" twice, and then "go up". Now "use key" and once inside the room, "go up". "Use flashlight" again, and then "pickup bat". Then you need to find the cook for yourself. Go around until you see him in a room, he won't move as fast as you. When you see him, "accuse cook bat". This will accuse him with the decisive evidence, and you will win the game.

Known bugs or problems

There are no known bugs or problems with the game.

K1925245


```
1
2  /**
3   * Item room - An item in an adventure game
4   *
5   * This class is part of "The murder of Melissa Roland"
6   * "The murder of Melissa Roland" is a text based adventure game
7   *
8   * An "Item" represents one of the various objects present in each room.
9     Some can be picked up, others can't.
10  * Items have weight, which prevents the player from picking up too many.
11  * Each item also has its own description, shown when the player examines the
    item
12  *
13  * The beacon is a particular item that uses two additional fields: the room it
    memorizes, and its activity status
14  *
15  * @author Christian Impollonia
16  * @version 2019.11.26
17  */
18
19 public class Item
20 {
21     // instance variables
22     private int weight;
23     private boolean canPickUp;
24     private String description;
25
26     //used for beacons, to memorize rooms and beacon status. Set as null because
    all other items will not be using them.
27     private Boolean active = null;
28     private Room savedRoom = null;
29
30     /**
31      * Constructor for items. The possibility to be picked up, the weight and the
    description are all parameters
32      */
33     public Item(int weight, boolean canPickUp, String description)
34     {
35         this.weight=weight;
36         this.canPickUp=canPickUp;
37         this.description=description;
38
39     }
40
41     /**
42      * Returns true if the item can be picked up, false if not.
43      *
44      */
45     public boolean getCanPickUp()
46     {
47         return canPickUp;
48     }
49
50     /**
```

K1925245


```
51      * Returns the weight of the item
52      */
53     public int getWeight()
54     {
55         return weight;
56     }
57
58     /**
59      * Prints out the description of the item
60      */
61     public void printDescription()
62     {
63         System.out.println(description);
64     }
65
66     /**
67      * Changes the status of the beacon, by assigning true or false to the
    isActive field
68      */
69     public void setActive(boolean active)
70     {
71         this.active = active;
72     }
73
74     /**
75      * Returns the active status of the beacon
76      */
77     public Boolean isActive()
78     {
79         return active;
80     }
81
82     /**
83      * Sets the saved room to a specific room
84      */
85     public void setRoom(Room room)
86     {
87         savedRoom = room;
88     }
89
90     /**
91      * Returns the room saved in the beacon
92      */
93     public Room getSavedRoom()
94     {
95         return savedRoom;
96     }
97 }
98
```

K1925245


```java
/**
 *  This class is the main class of "The murder of Melissa Roland".
 *  "The murder of Melissa Roland" is a text based adventure game where you are
     asked to find and arrest a murderer!
 *
 *  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 all the commands that the parser returns.
 *
 * @author  Michael Kölling, David J. Barnes and Christian Impollonia
 * @version 2019.11.26
 * K number 1925245
 */
import java.util.*;

public class Game
{
    private Parser parser;
    private Room entrance, lounge, kitchen, basement, hall, garden, bedroom,
attick, magicRoom;
    private Bag bag;
    private Character cook, butler, gardener;
    private Item chicken1, book1, radio1, baseballBat1; //declared outside the
createItems method because needed in the createCharacters and accuseCharacter
methods
    private MainPlayer player;
    private ArrayList<Room> rooms; //The list of rooms in the map the player or
the characters can teleport to



    /**
     * Create the game and initialise its internal map, by also placing
characters and items in their respective rooms.
     * Also creates the bag and places the player at the entrance.

     */
    public Game()
    {
        createRooms();
        createItems();
        createCharacters();
        player = new MainPlayer(entrance);
        parser = new Parser();
        bag = new Bag(50);                              //creates the player's bag
with a max weight of 50
        rooms = new ArrayList<>();
        fillRoomsList();
```

K1925245


```
48          player.setFirstTime(true);                  //considers that the
player will start the game for the first time
49
50
51      }
52
53      /**
54       * Create all the rooms and link their exits together.
55       */
56      private void createRooms()
57      {
58
59          // create the rooms
60          entrance = new Room("I'm in the main entrance of the Roland mansion",
false, false);
61          lounge = new Room("This is the lounge. Huge room but so quiet", false,
false);
62          kitchen = new Room("What a pleasant smell. I'm in the kitchen", false,
false);
63          basement = new Room("Now that's much better, this is the basement. Let's
see if I can find anything useful", true, false);
64          hall = new Room("The hall. Also known as the crime scene. I should
examine it well.", false, false);
65          garden = new Room("The back garden. It's still storming with rain.",
false, false);
66          bedroom = new Room("Ok, I'm finally in. This must be Melissa's bedroom",
false, true);
67          attick = new Room("What a small attick. I can feel this room holds a big
secret", true, false);
68          magicRoom = new Room("This doesn't feel like a normal room. \n *ZAP* \n
What's going on? \n I can't believe it, I just moved to a different room!",
false, false);
69
70          // initialise room exits
71          entrance.setExit("east", kitchen);
72          entrance.setExit("north", lounge);
73          entrance.setExit("west", hall);
74
75          lounge.setExit("south", entrance);
76          lounge.setExit("north", magicRoom);
77
78          kitchen.setExit("west", entrance);
79          kitchen.setExit("down" , basement);
80
81          basement.setExit("up" , kitchen);
82
83          hall.setExit("north", garden);
84          hall.setExit("east" , entrance);
85          hall.setExit("up" , bedroom);
86
87          bedroom.setExit("down" , hall);
88          bedroom.setExit("up" , attick);
89
90          attick.setExit("down" , bedroom);
```

K1925245


```
91
92          garden.setExit("south", hall);
93
94      }
95
96      /**
97       * Create all items and add them to each room
98       */
99      private void createItems()
100     {
101         Item door1, fireplace1, carpet1, table1, letter1,
102         key1, manuscript1, body1, knife1, flashlight1, window1, barn1, shovel1,
103         medicine1, bed1, tree1, photo1, plate1, painting1, bookshelf1, oven1,
umbrella1, stretcher1, beacon1;
104
105         //create the items
106         door1=new Item(0, false, "The huge door I just came in from. Not gonna
use it again until I finish my job");
107         fireplace1=new Item(0, false, "This enormous fireplace isn't really
making that much of a difference. It's still freezing in here");
108         carpet1=new Item(0, false, "A really beautiful carpet. You can see how
rich the owner of this place is");
109         table1=new Item(0, false, "You can see someone had dinner here
recently");
110         book1=new Item(10, true, "I don't know what a book is doing on the floor.
The title reads: 'Best ways to smash a pinata'.Don't think it'll be much use");
111         letter1=new Item(3, true, "There's a letter near the window. It looks
incomplete. It reads: 'Dear Grace, there's something I need to tell you, I just
can't keep it to myself anymore..'It ends there");
112         baseballBat1=new Item(40, true, "A baseball bat. It's covered in blood,
and...mayonnaise? This must be the real murder weapon. It all makes sense now. I
need to take this with me and show it to the man responsible. This ends now.");
113         key1=new Item(3, true, "A small key. I wonder what this opens. I should
definitely take it with me");
114         manuscript1=new Item(5, true, "An old manuscript about this house and its
first owners");
115         body1=new Item(0, false, "Melissa Roland. The victim. A knife is stuck in
her body but I think there's a lot more to this story. Her head is cracked open,
the knife must have been planted post mortem. Now who would do that, and why? And
where's the real murder weapon?");
116         knife1=new Item(10, true, "The murder weapon. Or at least that's what the
killer wants us to think. There's something off about it.");
117         radio1=new Item(10, true, "A small radio. Doesn't have any batteries in
it. Not very useful to me");
118         flashlight1=new Item(5, true, "A pretty old flashlight, but it seems like
it's still working. This can help me a lot to analyze those darker rooms");
119         window1=new Item(0, false, "A broken window. But it look like it's been
broken for a long time");
120         barn1=new Item(0, false, "An old wooden barn. Nothing to see though, it's
empty");
121         shovel1=new Item(40, true, "What's a shovel doing on the floor. I'm sure
someone is looking for it");
122         medicine1=new Item(5, true, "A small bottle of pills. I wonder who they
belong to.");
```

K1925245


```
123         bed1=new Item(0, false, "The victim's bed. Pretty huge for a single
    bed");
124         tree1=new Item(0, false, "The only tree in the garden. It's quite
    majestic");
125         chicken1=new Item(20, true, "A roast chicken. The chef said I can take
    it. I guess it can't hurt");
126         photo1=new Item(5, true, "A photo of Ms Roland. She was quite
    beautiful");
127         plate1=new Item(5, true, "The only plate on the table. It was obviously
    used yesterday");
128         painting1=new Item(0, false, "A portrait of the first owner of the
    mansion, Jeremy Roland");
129         bookshelf1=new Item(0, false, "A very nice bookshelf. It looks like one
    book is missing though");
130         oven1=new Item(0, false, "A gigantic oven, made of rock");
131         umbrella1=new Item(25, true, "A small umbrella. Not of much use to me in
    here");
132         stretcher1=new Item(5, true, "A strange metallic item. I think I can use
    it to stretch things and make them larger. It might work for my bag.");
133         beacon1=new Item(3, true, "A tiny device, it looks like a beacon of some
    sort. I think that if I use it in one room, it will help me find my way back to
    it");
134
135         //add each item to its corresponding room
136         entrance.addItem("door", door1);
137         entrance.addItem("fireplace", fireplace1);
138         entrance.addItem("carpet", carpet1);
139         lounge.addItem("table" , table1);
140         lounge.addItem("book" , book1);
141         bedroom.addItem("letter" , letter1);
142         attick.addItem("bat" , baseballBat1);
143         kitchen.addItem("oven" , oven1);
144         attick.addItem("manuscript" , manuscript1);
145         hall.addItem("body" , body1);
146         hall.addItem("knife" , knife1);
147         basement.addItem("radio" , radio1);
148         garden.addItem("flashlight" , flashlight1);
149         bedroom.addItem("window" , window1);
150         garden.addItem("barn" , barn1);
151         lounge.addItem("shovel" , shovel1);
152         kitchen.addItem("medicine" , medicine1);
153         bedroom.addItem("bed" , bed1);
154         garden.addItem("tree" , tree1);
155         kitchen.addItem("chicken" , chicken1);
156         attick.addItem("photo" , photo1);
157         lounge.addItem("plate" , plate1);
158         entrance.addItem("painting" , painting1);
159         hall.addItem("bookshelf" , bookshelf1);
160         basement.addItem("key" , key1);
161         basement.addItem("umbrella" , umbrella1);
162         bedroom.addItem("stretcher", stretcher1);
163         hall.addItem("beacon", beacon1);
164
165
```

K1925245


```
166     }
167
168     /**
169      * Create all characters and put them in their initial rooms
170      */
171     private void createCharacters()
172     {
173
174
175         //create all the characters
176         cook = new Character("cook" , "Hi, I'm Ollie, the cook of the mansion,
nice to meet you. While the murder happened, I was here in the kitchen making my
mayonnaise sauce. It couldn't have been me. I can't find my radio anywhere. Would
you happen to know where it is? I'll tell you something important if you find it
for me.", radio1, "Thank you for finding my radio. I heard that the knife that
killed the victim was the gardener's");
177         gardener = new Character("gardener" , "Hey, I'm the gardener here, name's
Steven. Don't bother suspecting me, I was in the garden all day pruning the
plants. That made me very hungry though. I can help you out if I get something to
eat", chicken1, "Thanks, this chicken is delicious. I'll tell you that someone
lost his flashlight in the garden. It might be of help to you");
178         butler = new Character("butler" , "Good evening sir, I'm Jeremiah, miss
Roland's butler. I'm really sorry about what happened, I had nothing to do with
it. I was reordering the the books in the library yesterday. I can't find one of
those books anymore. Can you help me find it? I'll help you with your
investigation if you do.", book1, "Thank you sir, I finally have it. Now about
the help I promised you, I noticed there is a small key in the basement, it
should open the bedroom");
179
180         //assign each character to its initial room
181
182         kitchen.addCharacter(cook);
183         garden.addCharacter(gardener);
184         lounge.addCharacter(butler);
185
186     }
187
188     /**
189      *  Main play routine.  Loops until end of play.
190      *  Prints the full welcome only if it is the first time the player starts
191         the game. In case it isn't, there is no need to print the whole
intruduction,
192         only the room description and help message are printed.
193      */
194     public void play()
195     {
196         if(player.isFirstTime())
197             printWelcome();
198         else
199         {
200          System.out.println(player.getCurrentRoom().getLongDescription());
201          System.out.println("Type help if you need any");
202         }
203
```

K1925245


```
204         // Enter the main command loop.  Here we repeatedly read commands and
205         // execute them until the game is over.
206
207         boolean finished = false;
208         while (! finished) {
209             Command command = parser.getCommand();
210             finished = processCommand(command);
211         }
212         System.out.println("Thank you for playing.  Good bye.");
213     }
214
215     /**
216      * Prints out the opening message for the player.
217      * Message only printed out when the player first starts the game
218      */
219     private void printWelcome()
220     {
221         System.out.println();
222         System.out.println("Welcome to The murder of Melissa Roland");
223         System.out.println("You are the main detective in a murder case.");
224         System.out.println("A mysterious woman named Melissa Roland has been
    killed in an old mansion.");
225         System.out.println("There has been a storm outside for the past 12 hours,
    so no one could have left the place.");
226         System.out.println("The murderer must be one of the people in the
    mansion.");
227         System.out.println("Your objective is to find concrete evidence to arrest
    whoever did it.");
228         System.out.println("Good luck.");
229         System.out.println(player.getCurrentRoom().getLongDescription());
230         System.out.println("You can start by typing 'help', to get to know the
    commands you can use");
231         player.setFirstTime(false);                                  //makes sure
    this message is only printed out the first time
232     }
233
234     /**
235      * Given a command, process (that is: execute) the command.
236      * Some commands can't be executed in locked or dark rooms, so it uses the
    checkRoom method to see the status of the room
237      * @param command The command to be processed.
238      * @return true If the command ends the game, false otherwise.
239      */
240     private boolean processCommand(Command command)
241     {
242         boolean wantToQuit = false;
243
244         if(command.isUnknown()) {
245             System.out.println("I don't know what you mean...");
246             return false;
247         }
248
249         String commandWord = command.getCommandWord();
250         if (commandWord.toLowerCase().equals("help")) {
```

K1925245


```
251             printHelp();
252         }
253         else if (commandWord.toLowerCase().equals("go")) {
254             if(player.getCurrentRoom().isLocked())
255                 System.out.println("This room is locked. There is nowhere for me
    to go but back");
256             else
257                 goRoom(command);
258         }
259         else if (commandWord.toLowerCase().equals("quit")) {
260             wantToQuit = quit(command);
261         }
262         else if (commandWord.toLowerCase().equals("pickup")) {
263            if(checkRoom())
264                return false;
265            pickUpItem(command);
266         }
267
268         else if (commandWord.toLowerCase().equals("drop"))
269         {
270            if(checkRoom())
271                return false;
272            dropItem(command);
273         }
274         else if (commandWord.toLowerCase().equals("examine"))
275         {
276            if(checkRoom())
277                return false;
278            examineItem(command);
279         }
280         else if (commandWord.toLowerCase().equals("back"))
281         {
282             goBack(command);
283         }
284         else if (commandWord.toLowerCase().equals("talk"))
285         {
286             if(checkRoom())
287                 return false;
288             talk(command);
289         }
290         else if(commandWord.toLowerCase().equals("use"))
291         {
292             useItem(command);
293         }
294         else if(commandWord.toLowerCase().equals("give"))
295         {
296             if(checkRoom())
297                 return false;
298             giveItem(command);
299         }
300         else if(commandWord.toLowerCase().equals("accuse"))
301         {
302             if(checkRoom())
303                    return false;
```

K1925245


```
304             wantToQuit = accuseCharacter(command);
305         }
306         else if(commandWord.toLowerCase().equals("bag"))
307         {
308             showBag(command);
309         }
310
311         // else command not recognised.
312
313         return wantToQuit;
314     }
315
316     /**
317      * This method checks whether a room is locked or dark when a player tries to
    use a command. If it is, it will block the
318        execution of the command and print out a message
319      * @return true if room is unusable, false if not
320      */
321     private boolean checkRoom()
322     {
323         boolean check = false;
324         if(player.getCurrentRoom().isLocked())
325             {
326                 System.out.println("I need to get in the room first!");
327                 check = true;
328
329             }
330         else if(player.getCurrentRoom().isDark())
331             {
332                 System.out.println("It's too dark. I can't see in here");
333                 check = true;
334
335             }
336         return check;
337     }
338
339     /**
340      * Moves one character from its room to a random room from the list of exits
341      * Characters are not allowed in the bedroom, basement and attick, that
    require special items to access them.
342      * They are also not allowed in the magic transporter room
343      */
344     private void moveCharacter(Character character)
345     {
346         ArrayList <String> exitsList;
347
348         exitsList = character.getCurrentRoom().getExits();
349         exitsList.remove("up");     //makes sure that characters can't go upstairs
    or downstairs, where they are not allowed
350         exitsList.remove("down");
351         if(character.getCurrentRoom() == lounge)
352                 exitsList.remove("north");          //makes sure that characters
    can't go to the magic room
353         Random randomizer = new Random();
```

K1925245


```
354         int index = randomizer.nextInt(exitsList.size());
355         character.getCurrentRoom().removeCharacter(character);
356 
    character.getCurrentRoom().getExit(exitsList.get(index)).addCharacter(character);
357 
358 
359     }
360 
361 
362     /**
363      * Fills the list of rooms with the rooms in the map, where the player can
    teleport to using the magic room
364      * The attick is the only excluded room, given that it is a room that is
    inside a locked room. If the player were able to teleport
365        there, he would be able to complete the game unnaturally.
366      */
367     private void fillRoomsList()
368     {
369         rooms.add(garden);
370         rooms.add(kitchen);
371         rooms.add(lounge);
372         rooms.add(entrance);
373         rooms.add(hall);
374         rooms.add(basement);
375         rooms.add(bedroom);
376     }
377 
378     /**
379      * Moves all com characters in the game
380      */
381     private void moveAllCharacters()
382     {
383         moveCharacter(gardener);
384         moveCharacter(cook);
385         moveCharacter(butler);
386     }
387 
388     /**
389      * This method is called every time the player moves. It counts how many
    times they have moved, and every specific amount
390        of times, it moves all characters.
391      * Right now it is set at 8
392      */
393     private void moveCharactersEvery()
394     {
395         player.increaseMovementCount();
396 
397         if(player.getMovementCount()==8)
398         {
399             moveAllCharacters();
400             player.resetMovementCount();
401         }
402     }
403 
```

K1925245


```
404     // implementations of user commands:
405
406     /**
407      * Print out some helpful information.
408      * Here we print a list of the
409      * command words
410      */
411     private void printHelp()
412     {
413         System.out.println("You are lost. We can help.");
414         System.out.println();
415         System.out.println("Your command words are:");
416         parser.showCommands();
417
418         }
419
420     /**
421      * Try to move in one direction. If there is an exit, enter the new
422      * room, otherwise print an error message.
423      * If the room the player is moving to is the magic room, it will randomly
    teleport the player to one of the accessible rooms
424      * It will also move all characters every 8 times this method is called
425      */
426     private void goRoom(Command command)
427     {
428         if(!command.hasSecondWord()) {
429             // if there is no second word, we don't know where to go...
430             System.out.println("Go where? Next time follow the go command with
    the name of the exit you want to use");
431             return;
432         }
433
434         String direction = command.getSecondWord();
435
436         // Try to leave current room.
437         Room nextRoom = player.getCurrentRoom().getExit(direction);
438
439         if (nextRoom == null) {
440             System.out.println("There is no door!");
441         }
442
443         else {
444             player.setPreviousRoom(player.getCurrentRoom());       //makes sure
    it remembers this room if I want to go back
445             if(nextRoom == magicRoom)
446             {
447                 System.out.println(nextRoom.getShortDescription());
448                 Random randomizer = new Random();
449                 player.setCurrentRoom(rooms.get(randomizer.nextInt(7)));
450              }
451             else
452                 player.setCurrentRoom(nextRoom);
453
454             moveCharactersEvery();
```

K1925245


```
455
456             player.getCurrentRoom().printRoomDescription() ;
457         }
458     }
459
460     /**
461      * "Quit" was entered. Check the rest of the command to see
462      * whether we really quit the game.
463      * @return true, if this command quits the game, false otherwise.
464      */
465     private boolean quit(Command command)
466     {
467         if(command.hasSecondWord()) {
468             System.out.println("Quit what? To quit the game, write the quit
command alone");
469             return false;
470         }
471         else {
472             return true;  // signal that we want to quit
473         }
474     }
475
476     /**
477      * This method processes the "pickup" command. First it checks if the second
command word has been inserted, or if the item
478        actually exists, if not it prints a message to the user.
479
480      * It will also tell the user if the is trying to pick up an object that
can't be picked up
481      *
482      * In the other case, it adds the item to the bag, but only if the weight of
the object is allowed, given the maximum weight
483        the bag can hold.
484
485      * If an item has been picked up, it will go inside the bag so it will be
removed from the room.
486      */
487     private void pickUpItem(Command command)
488     {
489         if(!command.hasSecondWord()) {
490             System.out.println("Pick up what? Next time follow the pickup command
with the name of an item in the room");
491             return;
492         }
493
494         String itemToPickUpKey = command.getSecondWord(); //gets the keyword for
the item we need
495         Item itemToPickUp = player.getCurrentRoom().getItem(itemToPickUpKey);
//gets the item itself using the keyword
496
497         if(itemToPickUp==null)
498         {
499             System.out.println("There is no such item!");
500         }
```

K1925245


```
501
502         else if(itemToPickUp.getCanPickUp())
503         {
504 if((bag.getCurrentWeight()+itemToPickUp.getWeight())<=bag.getMaxWeight())
505             {
506                 bag.addItem(itemToPickUpKey, itemToPickUp);
507                 player.getCurrentRoom().removeItem(itemToPickUpKey);
508                 System.out.println("Item added to the bag");
509             }
510             else
511                 System.out.println("Your bag is full! Free some space");
512         }
513         else System.out.println("You can't possibly pick that up!");
514
515
516     }
517
518     /**
519      * This method processes the "drop" command.
520      * First it checks if the second command word has been inserted, or if the
item
521        actually exists, if not it prints a message to the user.
522      * In the other case, it drops the item from the bag decreasing its current
weight.
523      * After being dropped, the item becomes part of the room the player is in at
that time.
524      */
525     private void dropItem(Command command)
526     {
527
528         if(!command.hasSecondWord())
529         {
530             System.out.println("Drop what? Next time follow the drop command with
the name of an item in the bag");
531             return;
532         }
533
534         String itemToDropKey = command.getSecondWord(); //gets the string for the
item we need
535         Item itemToDrop = bag.getItem(itemToDropKey); //gets the item itself
using the keyword
536
537         if(itemToDrop==null)
538
539             System.out.println("There is no such item in the bag!");
540
541         else
542         {
543             bag.removeItem(itemToDropKey);
544             player.getCurrentRoom().addItem(itemToDropKey, itemToDrop);
545             System.out.println("Item dropped");
546         }
547
```

K1925245


```
548
549
550
551
552     }
553
554     /**
555      * This method processes the "examine" command.
556      * First it checks if the second command word has been inserted, or if the
item
557        actually exists, if not it prints a message to the user.
558      * If conditions are met, it prints out the description of the item inserted
by the user.
559      *
560      */
561     private void examineItem(Command command)
562     {
563         if(!command.hasSecondWord())
564         {
565             System.out.println("Examine what? Next time follow the examine
command with the name of an item in the room");
566             return;
567         }
568
569         String itemToExamineKey = command.getSecondWord();
570         Item itemToExamine = player.getCurrentRoom().getItem(itemToExamineKey);
571
572         if(itemToExamine==null)
573             System.out.println("There is no such item");
574         else
575         {
576             itemToExamine.printDescription();
577         }
578
579     }
580
581     /**
582      * This method processes the "back" command.
583      * All it does is swap the values of the previousRoom field and the
currentRoom field.
584      * To do so a local variable temp is created.
585      * It counts as movement, so it will also move all characters when the
counter reaches the number 8.
586      */
587     private void goBack(Command command)
588     {
589         if(command.hasSecondWord())
590             {
591                 System.out.println("Back what? To go back, type the back command
alone");
592                 return;
593             }
594         Room temp;
595         temp = player.getPreviousRoom();
```

K1925245


```
596         player.setPreviousRoom(player.getCurrentRoom());
597         player.setCurrentRoom(temp);
598         moveCharactersEvery();
599         player.getCurrentRoom().printRoomDescription();
600     }
601
602     /**
603      * Talks to a character in that room, getting a response from him
604      * First it checks if the second command word has been inserted, or if the
    character
605        actually exists, if not it prints a message to the user.
606      * In the other case, it prints out the response of that character
607      */
608     private void talk(Command command)
609     {
610         if(!command.hasSecondWord())
611         {
612             System.out.println("Talk to who? Next time follow the talk command
    with the name of a character in the room");
613             return;
614         }
615
616         String characterKey = command.getSecondWord();
617         Character characterToTalkTo =
    player.getCurrentRoom().getCharacter(characterKey);
618
619         if(characterToTalkTo == null)
620         {
621             System.out.println("There's no such person!");
622         }
623         else
624         {
625             characterToTalkTo.printResponse();
626         }
627     }
628
629     /**
630      * This method lets the player use an item in his bag.
631      * First it checks if the second command word has been inserted, or if the
    item
632        actually exists, if not it prints a message to the user.
633      * Some items in the game are not very useful, and a message will be returned
    if
634        you try to use them.
635      * But if the player uses a key in a locked room or a flashlight in a dark
    room,
636        (if they are used in the wrong room a message will be displayed) then they
    will be able to explore it.
637      * They can also use the beacon and the stretcher.
638      */
639     private void useItem(Command command)
640     {
641         if(!command.hasSecondWord())
642         {
```

```
643             System.out.println("Use what? Next time, follow the use command with
    the name of an item in your bag");
644             return;
645         }
646
647         String itemKey = command.getSecondWord();
648         Item itemToUse = bag.getItem(itemKey);
649
650         if(itemToUse == null)
651         {
652             System.out.println("There is no such item in the bag!");
653
654         }
655
656         else if(itemToUse == bag.getItem("key"))
657              {
658                  if(player.getCurrentRoom().isLocked())
659                     {
660                         player.getCurrentRoom().unlockRoom();
661                         System.out.println("Room unlocked!");
662                         player.getCurrentRoom().printRoomDescription();
663                        }
664                  else
665                      System.out.println("This room is already unlocked");
666                    }
667         else if(itemToUse == bag.getItem("flashlight"))
668               {
669                   if(player.getCurrentRoom().isDark())
670                     {
671                         player.getCurrentRoom().brightenRoom();
672                         System.out.println("Let there be light!");
673                         player.getCurrentRoom().printRoomDescription();
674                        }
675                   else
676                       System.out.println("I can already see perfectly well in
    here");
677                    }
678         else if(itemToUse == bag.getItem("stretcher"))
679                 {
680                     bag.extend(20);
681                     System.out.println("Bag size was extended by 20!");
682                     System.out.println("Stretcher used and thrown away");
683                     bag.removeItem("stretcher");
684                 }
685         else if(itemToUse == bag.getItem("beacon"))
686                {
687                     Item beacon = bag.getItem("beacon");
688
689                     if(!beacon.isActive())
690                        {
691                             beacon.setRoom(player.getCurrentRoom());
692                             beacon.setActive(true);
693                             System.out.println("Room saved with the beacon. Use
    again to get to the room from anywhere");
```

K1925245


```
694                     return;
695                 }
696             if(beacon.isActive())
697                {
698                     player.setCurrentRoom(beacon.getSavedRoom());
699                     beacon.setActive(false);
700                     System.out.println("Beacon used!");
701                     player.getCurrentRoom().printRoomDescription();
702                     return;
703                 }
704
705              }
706         else
707                System.out.println("I can't find much use for this object");
708
709
710     }
711
712     /**
713      * This method lets the player give an item to a character in the room. If
    that character likes the item, he will give the
714        player an important clue. It is a 3-word command.
715      * First it checks if the second and third command words have been inserted,
    or if the item and the character
716        actually exist, if not it prints a message to the user.
717      * If the item is the happy item of the character that is receiving it, the
    character will become happy, changing his response.
718      */
719     private void giveItem(Command command)
720     {
721         if(!command.hasSecondWord())
722            {
723                 System.out.println("Give what? Next time follow the give command
    with the name of an item in your bag");
724                 return;
725             }
726         if(!command.hasThirdWord())
727            {
728                 System.out.println("Give it to who? Next time follow the name of
    the item with the name of a character in the room");
729                 return;
730             }
731
732         String itemToGiveKey = command.getSecondWord();
733         Item itemToGive = bag.getItem(itemToGiveKey);
734         String characterToReceiveKey = command.getThirdWord();
735         Character characterToReceive =
    player.getCurrentRoom().getCharacter(characterToReceiveKey);
736
737         if(itemToGive == null)
738              System.out.println("There's no such item in the bag!");
739         else if(characterToReceive == null)
740              System.out.println("There's no such character in the room!");
741
```

K1925245


```
742         else if(characterToReceive.getHappyItem() == bag.getItem(itemToGiveKey))
743                 {
744                     System.out.println("Item given away");
745                     characterToReceive.setNewResponse();
746                     characterToReceive.printResponse();
747                     bag.removeItem(itemToGiveKey);
748                   }
749         else
750                     System.out.println(characterToReceive.getName() + ": What
    am I supposed to do with that? Keep it");
751
752
753
754
755
756     }
757
758     /**
759      * This method implements the way to beat or lose the game: accusing a
    character. It works in a very similar way to the give command.
760      * The players follows the accuse command with the name of the character and
    then the name of the decisive evidence he is going to use.
761      * If the character is the killer and the evidence supports it, he wins the
    game. In any other case, he loses.
762      * @return true if the game is finished, false if it is not.
763      */
764     private boolean accuseCharacter(Command command)
765     {
766         if(!command.hasSecondWord())
767             {
768                 System.out.println("Accuse who? Next time follow the accuse
    command with the name of the person in the room you want to accuse of murder,
    followed by the item you think is decisive evidence. Don't get it wrong, or there
    will be consequences");
769                 return false;
770             }
771         if(!command.hasThirdWord())
772             {
773                 System.out.println("What is the decisive evidence? Follow the
    name of the accused with the evidence you think says he is guilty. Don't get it
    wrong, or there will be consequences");
774                 return false;
775             }
776
777         String characterToAccuseKey = command.getSecondWord();
778         Character characterToAccuse =
    player.getCurrentRoom().getCharacter(characterToAccuseKey);
779
780         String decisiveEvidenceKey = command.getThirdWord();
781         Item decisiveEvidence = bag.getItem(decisiveEvidenceKey);
782
783         if(characterToAccuse == null)
784             {
785                 System.out.println("There is no such person in this room");
```

K1925245


```
786                 return false;
787             }
788         else if(decisiveEvidence == null)
789             {
790                 System.out.println("There is no such item in the bag");
791                 return false;
792             }
793 
794         else if(characterToAccuse == cook && decisiveEvidence == baseballBat1)
795             {
796                 System.out.println("Correct! The cook, Ollie Dods, murdered
    Melissa and tried to hide the real murder weapon, the baseball bat! This is
    decisive evidence and he is being arrested on the spot.");
797                 System.out.println("Good job! You have won the game.");
798                 return true;
799 
800             }
801         else if(characterToAccuse == cook)
802             {
803                 System.out.println("You may have been on to something, but that
    evidence is just wrong. It doesn't prove anything.");
804                 System.out.println("You are charged with the crime of false
    accusation, and have been arrested on the spot");
805                 System.out.println("You lose :(");
806                 return true;
807 
808             }
809         else
810             {
811                 System.out.println("No, you got it all wrong, he has nothing to
    do with the crime! Poor choice.");
812                 System.out.println("You are charged with the crime of false
    accusation, and have been arrested on the spot");
813                 System.out.println("You lose :(");
814                 return true;
815 
816             }
817     }
818 
819     /**
820      * When called, this method prints out all items in the player's bag
821      */
822     private void showBag(Command command)
823     {
824         if(command.hasSecondWord())
825            {
826                 System.out.println("Bag what? To see the items in your bag, write
    the bag command alone");
827                 return;
828             }
829 
830         System.out.println(bag.itemsInBag());
831     }
832 
```

```
833
834
835 }
836
```

K1925245


```
1
2  /**
3   * Class Character - A class describing a character in an adventure game
4   *
5   * This class is part of "The murder of Melissa Roland"
6   * "The murder of Melissa Roland" is a text based adventure game.
7   *
8   * In this game, a character is a person roaming around the map.
9   * Characters are found in rooms, but differently from items, they can move
    around, between rooms.
10  *
11  * Character movements are scripted, not completely randomized. They are
    programmed to move every time the player changes room
12     a number of times. Right now that number is set to 8.
13  * Like the player, they can only move through an exit from the room they are in.
    They choose a random one to use.
14  *
15  * Characters have 5 main fields: a name, which is the word used to refer to
    them,
16     and a response, that is what they say when they are talked to.
17  * They memorize the current room they are in.
18  * They can also be made happy when a player gives them their happy item. When
    that happens, they start giving a happy response
19     instead of the normal response.
20  *
21  *
22  * @author Christian Impollonia
23  * @version 2019.11.26
24  */
25 public class Character
26 {
27     private String name;
28     private String response;
29     private Room currentRoom;
30     private Item happyItem;
31     private String happyResponse;
32
33     /**
34      * The constructor for Character objects
35      * The name of the character, the response, and both the happy item and happy
    response are all taken as parameters.
36      */
37     public Character(String name, String response, Item happyItem, String
    happyResponse)
38     {
39         this.name = name;
40         this.response = response;
41         this.happyItem = happyItem;
42         this.happyResponse = happyResponse;
43
44     }
45
46     /**
47      * Returns the name of the character
```

K1925245


```
48      */
49     public String getName()
50     {
51         return name;
52     }
53
54     /**
55      * Returns the response the character gives when talked to
56      */
57     public void printResponse()
58     {
59         System.out.println(name + ": " + response);
60     }
61
62     /**
63      * Sets what the current room of the character is
64      */
65     public void setCurrentRoom(Room room)
66     {
67         currentRoom = room;
68     }
69
70     /**
71      * Returns the current room the character is in
72      */
73     public Room getCurrentRoom()
74     {
75         return currentRoom;
76     }
77
78     /**
79      * Returns the happy item
80      */
81     public Item getHappyItem()
82     {
83         return happyItem;
84         }
85
86     /**
87      * Sets the happy response as the new response
88      */
89     public void setNewResponse()
90     {
91         response = happyResponse;
92     }
93
94
95
96
97
98
99 }
100
```

K1925245


```java
/**
 * This class is part of "The murder of Melissa Roland"
 * "The murder of Melissa Roland" is a text based adventure game.
 *
 * This class holds information about a command that was issued by the user.
 * A command currently consists of two strings: a command word and a second
 * word (for example, if the command was "take map", then the two strings
 * obviously are "take" and "map").
 *
 * 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 word is <null>.
 *
 * @author  Michael Kölling, David J. Barnes and Christian Impollonia
 * @version 2019.11.26
 */

public class Command
{
    private String commandWord;
    private String secondWord;
    private String thirdWord;

    /**
     * Create a command object. First and second word must be supplied, but
     * either one (or both) 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.
```

K1925245


```
55      */
56     public String getSecondWord()
57     {
58         return secondWord;
59     }
60
61     /**
62      * @return The third word of this command. Returns null if there was no third
    word
63      */
64     public String getThirdWord()
65     {
66         return thirdWord;
67     }
68
69     /**
70      * @return true if this command was not understood.
71      */
72     public boolean isUnknown()
73     {
74         return (commandWord == null);
75     }
76
77     /**
78      * @return true if the command has a second word.
79      */
80     public boolean hasSecondWord()
81     {
82         return (secondWord != null);
83     }
84
85     /**
86      * @return true if the command has a third word.
87      */
88     public boolean hasThirdWord()
89     {
90         return (thirdWord != null);
91     }
92 }
93
94
```

K1925245


```
1
2  /**
3   * Class Bag - The main character's bag in an adventure game
4   *
5   * This class is part of "The murder of Melissa Roland"
6   * "The murder of Melissa Roland" is a text based adventure game.
7   *
8   * A "Bag" represents the container in which the player will hold the items he
    picked up.
9   * It has a maximum weight it can hold, stated in the constructor and a current
    weight of the items it is holding.
10  * It also holds the list of the items it is holding.
11  *
12  * @author Christian Impollonia
13  * @version 2019.11.26
14  */
15
16 import java.util.HashMap;
17 import java.util.Set;
18
19 public class Bag
20 {
21
22     private int maxWeight;
23     private int currentWeight;
24     private HashMap<String, Item> itemsList;
25
26
27     /**
28      * Creates the itemList object and initializes the two weight variables,
    getting max weight as parameter
29      */
30     public Bag(int maxWeight)
31     {
32         this.maxWeight = maxWeight;
33         currentWeight = 0;
34         itemsList = new HashMap<>();
35     }
36
37     /**
38      * Adds item to the list of items in the bag
39      * Because every item has a certain weight, adds to the current weight of the
    bag
40      */
41     public void addItem(String keyword, Item item)
42     {
43         itemsList.put(keyword, item);
44         currentWeight = currentWeight + item.getWeight();
45
46     }
47
48     /**
49      * Returns the maximum weight the bag can hold
50      */
```

K1925245


```
51      public int getMaxWeight()
52      {
53          return maxWeight;
54      }
55
56      /**
57       * Returns the current weight of the bag
58       */
59      public int getCurrentWeight()
60      {
61          return currentWeight;
62      }
63
64      /**
65       * Returns a string describing the items in this bag
66       *
67       */
68      public String itemsInBag()
69      {
70          String returnString = "The items in your bag are: ";
71          Set <String> keys = itemsList.keySet();
72          for(String key : keys)
73          {
74              returnString = returnString + key + ", ";
75          }
76          return returnString;
77
78
79      }
80
81       /**
82       * Returns a specific item in the bag by using its keyword
83       */
84      public Item getItem(String keyword)
85      {
86          return itemsList.get(keyword);
87      }
88
89      /**
90       * Removes a specific item in the bag by using its keyword
91       * Because every item has a certain weight, subtracts from the current weight
    of the bag
92       */
93      public void removeItem(String keyword)
94      {
95          currentWeight = currentWeight - itemsList.get(keyword).getWeight();
96          itemsList.remove(keyword);
97      }
98
99      /**
100      * Extends the bag, changing the maximum weight of items it can carry
101      */
102     public void extend(int addedWeight)
103     {
```

K1925245


```
104         maxWeight = maxWeight + addedWeight;
105     }
106 }
107
```

K1925245


```
1
2  /**
3   * Class MainPlayer - A class describing the main player in an adventure game
4   *
5   * This class is part of "The murder of Melissa Roland"
6   * "The murder of Melissa Roland" is a text based adventure game.
7   *
8   * The main player is the character that is controlled by the user. He can move
    around the map, and has 4 main fields.
9   * The currentRoom field wich tells what room the player is currently in, the
    previousRoom field, that allows the user
10    to use the "back" command, and the firstTime field, that is true or false
    depending on whether the player has just started
11    the game, or is continuing from a previous save.
12  * The last field is the movement count, that counts how many times the player
    has moved from room to room
13
14  * @author Christian Impollonia
15  * @version 2019.11.26
16  */
17 public class MainPlayer
18 {
19
20     private Room currentRoom;
21     private Room previousRoom;
22     private boolean firstTime;     //checks whether it is the first time or not
23     private int movementCount;     //counts the amount of times the player
    changes rooms
24
25     /**
26      * Main player constructor, sets the initial room where the player begins the
    game, and makes sure that he remains in the same
27        room if the "back" command is typed in this first room.
28      */
29     public MainPlayer(Room room)
30     {
31         setCurrentRoom(room);
32         setPreviousRoom(room);
33     }
34
35     /**
36      * Sets a specific room as the player's current room
37      */
38     public void setCurrentRoom(Room room)
39     {
40         currentRoom = room;
41     }
42
43     /**
44      * Sets a specific room as the player's previous room
45      */
46     public void setPreviousRoom(Room room)
47     {
48         previousRoom = room;
```

K1925245


```
49      }
50
51      /**
52       * Returns the current room the player is in
53       */
54      public Room getCurrentRoom()
55      {
56          return currentRoom;
57      }
58
59      /**
60       * Returns the previous room the player was in
61       */
62      public Room getPreviousRoom()
63      {
64          return previousRoom;
65      }
66
67      /**
68       * Sets the first time field to a specific value
69       */
70      public void setFirstTime(boolean firstTime)
71      {
72          this.firstTime = firstTime;
73      }
74
75      /**
76       * Returns true if it is the first time the player has played or not, false
    otherwise
77       */
78      public boolean isFirstTime()
79      {
80          return firstTime;
81      }
82
83      /**
84       * Returns the amount of times the player has changed room
85       */
86      public int getMovementCount()
87      {
88          return movementCount;
89      }
90
91      /**
92       * Called when the player moves. Increases the movementCount by 1
93       */
94      public void increaseMovementCount()
95      {
96          movementCount++;
97      }
98
99      /**
100      * Resets the movementCount to 0
101      */
```

K1925245


```
102     public void resetMovementCount()
103     {
104         movementCount = 0;
105     }
106
107
108 }
109
```

K1925245


```
/**
 * This class is part of the "The murder of Melissa Roland"
 * "The murder of Melissa Roland" is a 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, David J. Barnes and Christian Impollonia
 * @version 2019.11.26
 */

public class CommandWords
{
    // a constant array that holds all valid command words
    private static final String[] validCommands = {
        "go", "quit", "help", "pickup", "drop", "examine", "back", "talk",
"use", "give", "accuse", "bag"
    };

    /**
     * Constructor - initialise the command words.
     */
    public CommandWords()
    {

    }

    /**
     * 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(aString!=null)
           {
            if(validCommands[i].equals(aString.toLowerCase()))
                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();
    }
```

```
54 }
55
```

K1925245


```
1  import java.util.*;
2
3  /**
4   * Class Room - a room in an adventure game.
5   *
6   * This class is part of "The murder of Melissa Roland"
7   * "The murder of Melissa Roland" is a text based adventure game.
8   *
9   * A "Room" represents one location in the scenery of the game.  It is
10  * connected to other rooms via exits.  For each existing exit, the room
11  * stores a reference to the neighboring room.
12  *
13  * Rooms also have their own description, and store items and characters in
    addition to exits.
14  *
15  * Some rooms can be temporarily locked or dark.
16  *
17  * @author  Michael Kölling, David J. Barnes and Christian Impollonia
18  * @version 2019.11.26
19  */
20
21 public class Room
22 {
23     private String description;
24     private HashMap<String, Room> exits;        // stores exits of this room.
25     private HashMap<String, Item> items;        // stores items of this room.
26     private HashMap<String, Character> characters;  //stores characters of this
    room.
27     private boolean dark;      //specifies if the room is dark or not
28     private boolean locked;    //specifies if the room is locked or not
29
30     /**
31      * Create a room described "description". Initially, it has
32      * no exits. "description" is something like "a kitchen" or
33      * "an open court yard".
34      * It is also given as a parameter if the room is dark, or locked
35      */
36     public Room(String description, boolean dark, boolean locked)
37     {
38         this.description = description;
39         exits = new HashMap<>();
40         items = new HashMap<>();
41         characters = new HashMap<>();
42         this.dark = dark;
43         this.locked = locked;
44     }
45
46     /**
47      * Define an exit from this room.
48      * @param direction The direction of the exit.
49      * @param neighbor  The room to which the exit leads.
50      */
51     public void setExit(String direction, Room neighbor)
52     {
```

K1925245


```
53          exits.put(direction, neighbor);
54      }
55
56      /**
57       * @return The short description of the room
58       * (the one that was defined in the constructor).
59       */
60      public String getShortDescription()
61      {
62          return description;
63      }
64
65      /**
66       * Return a description of the room in the form:
67       *     You are in the kitchen.
68       *     Exits: north, west,
69       *     Items: hat, cd,
70       *     Characters: john, fred,
71       * @return A long description of this room
72       */
73      public String getLongDescription()
74      {
75          return description + ".\n" + getExitString() + "\n" + getItemsString() +
    "\n" + getCharactersString();
76      }
77
78      /**
79       * Return a string describing the room's exits, for example
80       * "Exits: north west".
81       * @return Details of the room's exits.
82       */
83      private String getExitString()
84      {
85          String returnString = "Exits: ";
86          Set<String> keys = exits.keySet();
87          for(String exit : keys) {
88              returnString += exit + ", ";
89          }
90          return returnString;
91      }
92
93      /**
94       * Return the room that is reached if we go from this room in direction
95       * "direction". If there is no room in that direction, return null.
96       * @param direction The exit's direction.
97       * @return The room in the given direction.
98       */
99      public Room getExit(String direction)
100     {
101         return exits.get(direction);
102     }
103
104     /**
105      * Add an Item to this room
```

K1925245


```
106      */
107     public void addItem(String keyword, Item item)
108     {
109         items.put(keyword, item);
110     }
111
112     /**
113      * Returns the item with the specified keyword
114      */
115     public Item getItem(String keyword)
116     {
117         return items.get(keyword);
118     }
119
120     /**
121      * Returns the string list of items in this room
122      */
123     private String getItemsString()
124     {
125         String returnString = "Items in the room: ";
126         Set <String> keys = items.keySet();
127         for(String key : keys)
128         {
129             returnString = returnString + key + ", ";
130         }
131         return returnString;
132     }
133
134     /**
135      * Removes the item with the specified key from the list of items in the room
136      */
137     public void removeItem(String keyword)
138     {
139         items.remove(keyword);
140     }
141
142     /**
143      * Add character to list of characters in the room
144      */
145     public void addCharacter(Character character)
146     {
147         characters.put(character.getName(), character);
148         character.setCurrentRoom(this);
149     }
150
151     /**
152      * Returns the String list of characters in this room
153      */
154     private String getCharactersString()
155     {
156         String returnString = "Characters in this room: ";
157         Set <String> keys = characters.keySet();
158         for(String key : keys)
159         {
```

K1925245


```
160             returnString = returnString + key + ", ";
161         }
162         return returnString;
163     }
164
165     /**
166      * Removes a character from the list of characters in the room
167      */
168     public void removeCharacter(Character character)
169     {
170         characters.remove(character.getName());
171     }
172
173     /**
174      * Returns the character from the list of characters in the room using the
keyword associated to it
175      */
176     public Character getCharacter(String keyword)
177     {
178         return characters.get(keyword);
179     }
180
181     /**
182      * Unlocks the room by assigning false to the locked field
183      */
184     public void unlockRoom()
185     {
186         locked = false;
187     }
188
189     /**
190      * Makes the room usable by assigning false to the dark field
191      */
192     public void brightenRoom()
193     {
194         dark = false;
195     }
196
197     /**
198      * Returns true if the room is locked, false if it is not
199      */
200     public boolean isLocked()
201     {
202         return locked;
203     }
204
205     /**
206      * Returns true if the room is dark, false if it is not
207      */
208     public boolean isDark()
209     {
210         return dark;
211     }
212
```

K1925245


```
213     /**
214      * Returns an array with all the exits in this room
215      * The keyset from the hashmap is created, and then the set is converted to
an ArrayList of String
216      * This way the random generator can be used to choose a random exit to use
217     */
218     public ArrayList<String> getExits()
219     {
220         Set <String> exitsSet = exits.keySet();
221         ArrayList<String> exitsList;
222         exitsList = new ArrayList<>();
223         for(String exit : exitsSet)
224             {
225                 exitsList.add(exit);
226             }
227         return exitsList;
228
229
230
231     }
232
233     /**
234      * Checks wether the current room is locked or dark, if so it prints that
description.
235      * In any other case, it prints the description of the room
236      */
237     public void printRoomDescription()
238     {
239         //checks if the room is dark or locked
240             if(locked)
241                    System.out.println("I'm in front of the door to the room, but
it is locked. I need a key to open it. I should go back if I don't have it.");
242             else if(dark)
243                    System.out.println("It's completely dark in here. I can't do
anything. I believe a flashlight would help");
244
245             else
246                    System.out.println(getLongDescription());
247     }
248
249 }
250
251
```

K1925245


```java
import java.util.Scanner;

/**
 * This class is part of "The murder of Melissa Roland".
 * "The murder of Melissa Roland" is a 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 one, two or 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, David J. Barnes and Christian Impollonia
 * @version 2019.11.26
 */
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()) {
```

K1925245


```
54                  word3 = tokenizer.next();// get third word
55              }
56              // note: we just ignore the rest of the input line.
57          }
58      }
59
60      // Now check whether this word is known. If so, create a command
61      // with it. If not, create a "null" command (for unknown command).
62      if(commands.isCommand(word1)) {
63          return new Command(word1, word2, word3);
64      }
65      else {
66          return new Command(null, word2, word3);
67      }
68  }
69
70  /**
71   * Print out a list of valid command words.
72   */
73  public void showCommands()
74  {
75      commands.showAll();
76  }
77  }
78
```

README.TXT 2019-nov-26 19:06 Page 1

```
Project: "The murder of Melissa Roland"
Authors: Michael Kölling, David J. Barnes and Christian Impollonia(K1925245)

To start this application, create an instance of class "Game" and call its
"play" method.


User level description:
This is a text-based adventure game, in which you can do several things using
commands you type.
You can move around rooms, study and pick up objects you find and take them with
you.
You can also interact with characters, by talking to them or by giving them
objects you found.
The rooms you find are not always accessible, and your objective is to find out
what happened inside the mansion, and find the killer by finding concrete
evidence.

Implementation description:
The game was implemented by using 9 total classes, with the 'Game' class being
the main one.
The most significant parts of the game, like characters, items and rooms, all
have their own classes.
All these objects are created inside the constructor of the main class, when the
game is started.
Being text-based, the game revolves mainly around the use of HashMaps, that let
every one of the created objects be linked to Strings, obtained from player input
using the Scanner class.
With everything the user does, these objects will be passing around various maps
throughout the game, changing what the player can interact with.

```

K1925245