Mansion escape

v 1.0

"Mansion escape " is a text-based command game, which can be played by inputting commands and reading corresponding responses for each and every one of them. As most other games "Mansion escape" , true to its title places the player in an environment where they have to complete a certain amount of tasks to escape. Each successful completion of a task awards the player with a piece of the key. Once all the pieces have been collected the game ends and the player has won.

The game starts in the starter room – where you wake up. You are then given a description of the situation as a form of intro to the game. After it you are free to explore the rooms to get yourself familiar with the mansion itself.
Among the rooms there are bosses/characters which would prompt you to complete challenges given by them every time you enter a room.
The layout of the mansion is as follows: there is one room for each direction of the world – respectively:

- north room – dining room
- north-east room- the north-east storage room
- east room – library
- south-east room- the south-east bedroom
- south room – the starter room
- south-west room – the south-west bedroom
- west room – a stairwell
- north-west room – a storage room

Other rooms include: the two levels below and above the stairwell – a tower and a dungeon and a transporter room – east of the library.
The player has to wander in these rooms in search of the key fragments. There are three main bosses guarding the keys. There are also randomly moving characters which may or may not be useful for the player. Upon successfully completing a challenge/task/quiz the player is presented with a third of the key! There are secrets left for the player to discover which are all to his benefit.

Task *completion*

Base tasks:

- The game has several locations/rooms.

The functionality of the rooms and the movement between them was already implemented in the code received. Therefore adding rooms was no difficult task. The above-mentioned rooms were all added with the room.getExit(direction,roomToGoTo) method. This is possible thanks to the rooms HashMap in the Room class.

- The player can walk through the locations. (This is already implemented in the code you are given).

Indeed this functionality too had already been implemented. Thanks to the goRoom() method the player is made able to type go and add a direction to move to a correspondingly mapped room. The method itself utilises a private variable currentRoom and another one-nextRoom. Next result is the result of the method call from the Room class – getExit(direction) where the direction is determined with the help of the parser being the second word of the user input. A few checks are validating if the input is valid or not and according messages are being printed in case of invalid input. Once the next room has been gotten

by the HashMap in the Room class the room's description set in the constructor of the Room class is being printed out and thus the player is being moved to the next chosen word.

- 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.

Initially for the completion of this base task I considered creating a separate Item class. Then it occurred to me that because the items in the game had nothing but a name as a string and a weight – an integer creating such a class would prove useless. Hence items became a HashMap in the Room class where the name of the item was the key and the weight – the value of the item, mapped to each key. I then created a setter method in the Room class for allocating items to each room in the game class. I also implemented a method in the Room class to return to the game class a string of all the items in the room following the 'inspect command'. What I also implemented was a method in the Room class to return to the game class the weight of an item(an integer). Now it was is simply time to put items in the rooms. This happened in the createRooms method of the Game class.
Implementing the ability to take and drop items required the use of two additional commands- a take and a drop command and a special ArrayList holding the items called inventory with a total storage space of 30 kg. Which brings us to the next task!

- 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.

Here all the magic for both this and the previous tasks happens in the take and drop command. The take command implements the getItem() method. There I add checks to validate whether the item is in fact an item and if I have enough room in my inventory to add it. That is where the getItemWeight() method comes into play. The dropping of items happens in a similar way but this time all the checks are in the inventory. Both tasks are completed in these two methods.

- 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.

A victory is achieved when the player collects the third and/or final piece of the key. Throughout the course of the games upon a successful quest a boolean is checked. Once three such have been set to true the game ends and a victory message is printed on the screen. There are more than one ways to collect the fragments. Order doesn't matter! They can be collected only once.

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

Achieved by implementing an ArrayList of rooms to keep track of all the rooms the player has been in. A counter variable comes to help us as well. With each command call if the command is not the back command it is being reset. That way we deal with consecutive calls of the back command by increasing it only

when the back command is called. Then simply getting the last element of the array according to the counter variable.

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

The total amount of commands I am using in my games is 15.

Challenge tasks:

- Add characters to your game. Characters are people or animals or monsters – anything that moves, really. Characters are also in rooms (like the player and the items). Unlike items, characters can move around by themselves.

For this purpose I have created an npc class. Npc are allowed to move randomly around the map. If you encounter them something may or may not happen depending on the npc. They each start off in a random room and with each move of the player they choose a random exit of this room and they go to the room mapped to it. A characteristic of the npc is that it can die.

- Extend the parser to recognise three-word commands. You could, for example, have a command `give bread dwarf` to give some bread (which you are carrying) to the dwarf.

This command had been implemented after making changes to the Command and Parser classes. In fact, winning the game is impossible without the use of the three-word command. It is crucial.

- Add a magic transporter room – every time you enter it you are transported to a random room in your game.

Situated east of the library, hidden amongst bookshelves is in fact the magical transporter room. To make it possible I first created an array to iterate over of all the rooms in the game. Then I created an instance of the random class and implemented a random index which can go as high as the size of the array with rooms -1. Then I just returned the description of the room at this index and setting it the current room.

- Others. You can invent additional challenge tasks yourself.

As far as personal challenges went I wanted to make the game as interesting as possible with my current time and ability. Therefore, I hid a character somewhere in the game which would give the player a side quest and as a reward they will receive accurate information as to how to win the game.

I also gave a back-up key fragment to one of the NPC characters. In case the player can not find the fragments guarded by the bosses he can resort to interaction with the NPC characters.

Code quality considerations:

- Coupling I considered when I created classes and used the ones already given . I made sure no two classes were using each other.(NPC class)

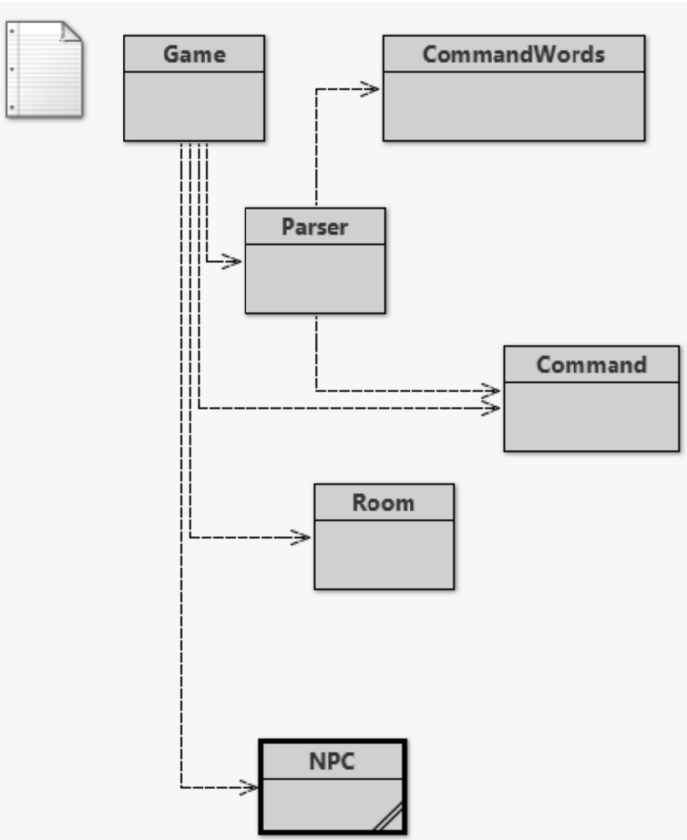
- Cohesion. To avoid high cohesion and make my code as maintainable and easy to read and re-use as possible I implemented methods wherever I could. (getRandomNeighbouringRoom()) Although seemingly complex at first, the methods in the classes are easy to read and understand as they are very logically connected.
- Responsibility-driven design. Wherever necessary classes were created. Wherever necessary – methods. No method does two or more different tasks.

Classic Walkthrough:

Commands go as follows:
read note
help
go north
inspect
go north
go south
go south
inspect
go east
inspect
back
go west
inspect
take recipebook
back
go north
go north
give recipebook chef – 1/3 key
go west
inspect
go east
go south
go east
reply 3 -1/3 key
go west
go west
go down
go up
go east
go north
go west
take meat
back
back
go west
go down
give meat monster -1/3 key
WIN!

There are more than 1 ways to win the game!

Known bug: if you swap second and third word of the give -item-character command nothing happens. Press enter to continue.


Game
CommandWords
Parser
Command
Room
NPC

```
1
2   /**
3    * NPC class
4    * This class is part of the "Mansion escape" game.
5    * "Mansion escape" is a very simple, text based adventure game.
6    *
7    * This class creates npc-objects. They have descriptions.
8    * @author  Michael Kölling and David J. Barnes and Mihail Bratanov
9    * @version 1.0
10   */
11  public class NPC
12  {
13      //description of the npc
14      private String description;
15      private boolean isAlive;
16
17      /**
18       * Constructor for objects of class NPC
19       */
20      public NPC(String description)
21      {
22          // initialise description
23          this.description=description;
24          isAlive = true;
25      }
26
27      /**
28       * Gets npc's description and returns it to other classes to be used further.
29       */
30      public String getDescription()
31      {
32          return description;
33      }
34       /**
35       * Sets npc to either dead or alive.
36       */
37      public void setAlive(boolean aliveChange)
38      {
39          isAlive = aliveChange;
40
41      }
42       /**
43       * Returns npc's state of life :D
44       */
45       public boolean getAlive()
46      {
47          return isAlive;
48      }
49  }
50
```

```
import java.util.*;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Random;
import java.util.ArrayList;
/**
 *
 *  This class is the main class of the "Mansion escape" game.
 *  "Mansion escape" is a classic escape game where the player is put in
 *  a situation where they need to complete a certain set of tasks to be able to escape.
 *
 *  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 and Mihail Bratanov
 * @version 1.0
 */

public class Game
{
    private Parser parser;
    private Room currentRoom;
    private Room previousRoom;
    private ArrayList<Room> visitedRooms;//all the rooms the player has visited go here
    private ArrayList<String>inventory;//any picked up item would be addded here
    private int backStepCounter=0;
    private int storageSpace=30; //total storage space of the inventory
    private int currentStorageSpace=0;
    private int spaceLeft=0;
    private boolean hasKey1=false;
    private boolean hasKey2=false;
    private boolean hasKey3=false;
    private boolean hasSpare=false;
    private ArrayList<Room> roomsToBeRandomlyChosen;//arrayList of Room typed objects to be choosen randomly
    private  Room starterRoom, library, mainHall, diningRoom, stairwell,northWestStorageRoom,northEastStorageRoom,
southWestBedroom, southEastBedroom, transporter, tower, dungeonGrounds;
    private NPC winnie;
    private NPC badWinnie;
    /**
     * Create the game and initialise its internal map.
     */
    public Game()
    {

        roomsToBeRandomlyChosen=new ArrayList<Room>();
        createRooms();
        visitedRooms=new ArrayList<Room>();
        inventory=new ArrayList<String>();
        parser = new Parser();

    }

    /**
     * Create all the rooms and link their exits together.
     * Fill the rooms with items. Each item has a certain weight.
     * Adds non-playable characters as well. (npc's)
     */
    private void createRooms()
    {
        //create npc
        winnie=new NPC("___________________________________________________________________________________________\n| I
AM A WINNIE.I AM NOT GOING TO HELP YOU AT ALL BUT I AM HERE FOR MORAL
SUPPORT!|\n__________________________________________________________________________________________\n");
        badWinnie=new
NPC("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n| I AM A  BAD
WINNIE.KILL ONE GOOD WINNIE AND I WILL GIVE YOU A SPARE PART OF THE
KEY!|\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
        // create the rooms
        transporter=new Room("being transported. Type in go to confirm!");
        starterRoom = new Room("where you wake up. Take a look around...");
        library = new Room("in the library. You are not alone here. There is a Librarian. They tell that if you
want to leave you need to answer the riddle correctly.\n It states:\n When is a door not a door? Your choices are:
\n 1.When it's brown!\n 2.When it is made of air!.\n 3.When it is open!\n Insert reply and number from 1 to 3 to
answer.");
```

```
70          mainHall = new Room("in the main hall");
71          diningRoom = new Room("in a dining room. There is a stranger sitting on the chair next to you. He is
    wearing a chef's hat. He is a chef! He tells you:\n Traveler if you want to escape this place I will give you a
    part of the key. But find and return me my recipe book! It is somewhere SOUTH of here! ");
72          stairwell = new Room("in a stairwell");
73          northWestStorageRoom=new Room("in the north-west storage room");
74          northEastStorageRoom=new Room("in the north-east storage room");
75          southWestBedroom=new Room("in the south-west bedroom");
76          southEastBedroom=new Room("in the south-east bedroom");
77          tower=new Room("in a mysterious tower.You suddenly hear a voice and turn around. There stands a wizzard
    just like the ones in Harry Potter! Only this one is missing his hat!\nHelp me find my hat traveler!\nIf you do I
    will provide you with all answers you need to escape this prison!" );
78          dungeonGrounds=new Room("in a dark dungeon. You're not alone here. Your eyes slowly adjust to the darkness
    and you find out you're standing face to face with a scary monster!!!\n^^^^^\n<0.0>\n<vvv>\n<^^^>\nvvvvvv");
79
80          // initialise room exits
81
82          starterRoom.setExit("north", mainHall);
83          starterRoom.setExit("east", southEastBedroom);
84          starterRoom.setExit("west", southWestBedroom);
85
86          southEastBedroom.setExit("north-west",mainHall);
87          southEastBedroom.setExit("west",starterRoom);
88
89          southWestBedroom.setExit("north-east",mainHall);
90          southWestBedroom.setExit("east",starterRoom);
91
92          mainHall.setExit("north", diningRoom);
93          mainHall.setExit("north-east", diningRoom);
94          mainHall.setExit("east", library);
95          mainHall.setExit("south-east", southEastBedroom);
96          mainHall.setExit("south", starterRoom);
97          mainHall.setExit("south-west", southWestBedroom);
98          mainHall.setExit("west", stairwell);
99          mainHall.setExit("north-west", northWestStorageRoom);
100
101         diningRoom.setExit("south", mainHall);
102         diningRoom.setExit("east", northEastStorageRoom);
103         diningRoom.setExit("west", northWestStorageRoom);
104
105         northEastStorageRoom.setExit("south-west",mainHall);
106         northEastStorageRoom.setExit("west",diningRoom);
107
108         northWestStorageRoom.setExit("south-east",mainHall);
109         northWestStorageRoom.setExit("east",diningRoom);
110
111         library.setExit("west" ,mainHall);
112         library.setExit("east",transporter);
113
114         stairwell.setExit("up",tower);
115         stairwell.setExit("east",mainHall);
116         stairwell.setExit("down",dungeonGrounds);
117
118         tower.setExit("down",stairwell);
119
120         dungeonGrounds.setExit("up",stairwell);
121
122         currentRoom = starterRoom;  // start game outside
123         //fill rooms with items
124         diningRoom.setItem("apple" , 5);
125         diningRoom.setItem("candle" , 3);
126         diningRoom.setItem("painting" , 8);
127         diningRoom.setItem("chair" , 10);
128         diningRoom.setItem("table",30);
129         northWestStorageRoom.setItem("wizzardHat",10);
130         northWestStorageRoom.setItem("meat",10);
131         northWestStorageRoom.setItem("beans",2);
132         northWestStorageRoom.setItem("eggplant",2);
133         northWestStorageRoom.setItem("peppers",2);
134         mainHall.setItem("piano",50);
135         mainHall.setItem("painting",50);
136         mainHall.setItem("lantern",20);
137         southWestBedroom.setItem("recipebook",10);
138         library.setItem("book",5);
139         library.setItem("bookshelf",500);
140         //prepare the rooms for the transporter
141         roomsToBeRandomlyChosen.add(starterRoom);
142         roomsToBeRandomlyChosen.add(library);
```

```
143         roomsToBeRandomlyChosen.add(mainHall);
144         roomsToBeRandomlyChosen.add(diningRoom);
145         roomsToBeRandomlyChosen.add(stairwell);
146         roomsToBeRandomlyChosen.add(northWestStorageRoom);
147         roomsToBeRandomlyChosen.add(northEastStorageRoom);
148         roomsToBeRandomlyChosen.add(southWestBedroom);
149         roomsToBeRandomlyChosen.add(southEastBedroom);
150         roomsToBeRandomlyChosen.add(tower);
151         roomsToBeRandomlyChosen.add(dungeonGrounds);
152         roomsToBeRandomlyChosen.add(transporter);
153 
154     }
155 
156     /**
157      *  Main play routine.  Loops until end of game.
158      *  If player has all the 3 parts of the key he will win the game.
159      */
160     public void play()
161     {
162         printWelcome();
163 
164         // Enter the main command loop.  Here we repeatedly read commands and
165         // execute them until the game is over.
166 
167         boolean finished = false;
168         while (! finished)
169         {
170             Command command = parser.getCommand();
171             finished = processCommand(command);
172             if((hasKey1==true && hasKey2==true & hasKey3==true ) || (hasSpare==true && hasKey2==true &
    hasKey3==true ) || (hasKey1==true && hasSpare==true & hasKey3==true ) || (hasKey1==true && hasKey2==true &
    hasSpare==true ))
173             {
174                 System.out.println("\n\nCONGRATULATIONS! YOU ESCAPED! YOU WON!\n\n");
175                 break;
176             }
177         }
178         System.out.println("Thank you for playing.  Good bye.");
179     }
180 
181     /**
182      * Print out the opening message for the player.
183      */
184     private void printWelcome()
185     {
186         System.out.println();
187         System.out.println("You wake up in a dark room. It is cold. The smell of cold stone and old furniture
    fills the room.\n A faint light!.. There is a candle on the cabinet next to you. A note is underneath it.\n Type
    'read note' to proceed...");
188         System.out.println();
189         //System.out.println(currentRoom.getLongDescription());
190 
191     }
192 
193     /**
194      * Given a command, process (that is: execute) the command.
195      * @param command The command to be processed.
196      * @return true If the command ends the game, false otherwise.
197      */
198     private boolean processCommand(Command command)
199     {
200         boolean wantToQuit = false;
201 
202         if(command.isUnknown())
203         {
204             System.out.println("I don't know what you mean...");
205             return false;
206         }
207 
208         String commandWord = command.getCommandWord();
209         if (commandWord.equals("help"))
210         {
211             backStepCounter=0;
212             printHelp();
213         }
214         else if (commandWord.equals("go"))
215         {
216             backStepCounter=0;
```

```
217             goRoom(command,roomsToBeRandomlyChosen);
218         }
219         else if (commandWord.equals("quit"))
220         {
221             backStepCounter=0;
222             wantToQuit = quit(command);
223         }
224         else if(commandWord.equals("read"))
225         {
226             backStepCounter=0;
227             read(command);
228         }
229         else if(commandWord.equals("get"))
230         {
231             backStepCounter=0;
232             get(command);
233         }
234         else if(commandWord.equals("inspect"))
235         {
236             backStepCounter=0;
237             getItems(command,inventory);
238         }
239         else if(commandWord.equals("back"))
240         {
241             backStepCounter++;
242             goBackRoom(backStepCounter);
243         }
244         else if(commandWord.equals("take"))
245         {
246             backStepCounter=0;
247             take(command,inventory);
248         }
249         else if(commandWord.equals("drop"))
250         {
251             backStepCounter=0;
252             drop(command,inventory);
253         }
254         else if(commandWord.equals("give"))
255         {
256             backStepCounter=0;
257             give(command,inventory);
258         }
259         else if(commandWord.equals("reply"))
260         {
261             backStepCounter=0;
262             reply(command);
263         }
264         else if(commandWord.equals("print"))
265         {
266             backStepCounter=0;
267             print(command);
268         }
269         else if(commandWord.equals("kill"))
270         {
271             backStepCounter=0;
272             kill(command);
273         }
274         // else command not recognised.
275         return wantToQuit;
276     }
277 
278     // implementations of user commands:
279     /**
280      *Kill command to kill winnnie and get the spareKey.
281      */
282     private void kill(Command command)
283     {
284         if(!command.hasSecondWord())
285         {
286             System.out.println("Select target to kill!");
287             return;
288         }
289         String secondString=command.getSecondWord();
290         if(secondString.toLowerCase().equals("winnie"))
291         {
292             System.out.println("Congratulations! YOU  murdered a pure and innocent Winnie! Welcome to the dark side! Here is your reward!\n You got the Spare key!!\n");
293             winnie.setAlive(false);
```

```
294             hasSpare=true;
295             return;
296 
297         }
298     }
299 
300     /**
301      *Print + location. Gives the player information about the current room he is in.
302      */
303     private void print (Command command)
304     {
305         if(!command.hasSecondWord())
306         {
307             System.out.println("Please specify what you wish to print out! Perhaps 'location'?");
308         }
309         String secondWord=command.getSecondWord();
310         if(secondWord.toLowerCase().equals("location"))
311         {
312             System.out.println(currentRoom.getLongDescription());
313         }
314     }
315 
316     /**
317      *Reply 1/2/3 - a command to answer the Librarian's question.
318      */
319     private void reply(Command command)
320     {
321         if(!command.hasSecondWord())
322         {
323             System.out.println("Please select one of the answers!\n");
324             return;
325         }
326         String secondString=command.getSecondWord();
327         if(secondString.startsWith("1"))
328         {
329             System.out.println("Wrong answer.Please try again!\n");
330             return;
331         }
332         else if(secondString.startsWith("2"))
333         {
334             System.out.println("Wrong answer.Please try again!\n");
335             return;
336         }
337         else if(secondString.startsWith("3"))
338         {
339             System.out.println("Correct! Here is your award stranger.\n You got 1/3 of the key.");
340             hasKey1=true;
341             return;
342         }
343 
344     }
345 
346     /**
347      * Give item character command- a 3-word command which gives the character an item from the inventory.
348      *GIVE ITEM from inventory to CHEF
349      *GIVE ITEM from invertory to MONSTER etc...
350      */
351     private void give(Command command, ArrayList inventory)
352     {
353         //implement cmd
354         if(!command.hasSecondWord())
355         {
356             // if there is no second word, we don't know where to go...
357             System.out.println("Get what? Perhaps you mean 'exits'...\n");
358             return;
359         }
360         String secondWord=command.getSecondWord();
361         if( inventory.contains(secondWord))
362         {
363             if(!command.hasThirdWord())
364             {
365                 System.out.println("Give item to whom? Perhaps you mean chef/monster/someone else?\n");
366                 return;
367             }
368 
369             String thirdWord=command.getThirdWord();
370             if(thirdWord.equals("chef"))
371             {
```

```
372                 System.out.println("Thank you kindly traveler!Here is your award.\n You got 1/3 of the key.\n");
373                 inventory.remove(command.getSecondWord());
374                 hasKey2=true;
375                 return;
376 
377             }
378             if(thirdWord.toLowerCase().equals("winnie"))
379             {
380                 System.out.println("Remember - Winnie is useless! DO not give items to Winnie. It is only here for moral support!");
381                 return;
382             }
383 
384             else if(thirdWord.equals("monster"))
385             {
386                 System.out.println(" ____");
387                 System.out.println("|@ @|");
388                 System.out.println("|VVV|");
389                 System.out.println("|^^^|");
390                 System.out.println(" ----");
391                 System.out.println("You got 1/3 of the key.\n");
392                 inventory.remove(command.getSecondWord());
393                 hasKey3=true;
394                 return;
395 
396             }
397             else if(thirdWord.equals("wizzard"))
398             {
399                 System.out.println("Thank you kindly traveler!Here is your award.\n The key once whole is now separated in three parts guarded by a chef in the dining room,\n a cruel librarian and a monster, residing in the dungeon bellow us.\nShould you wish to acquire the pieces take the recipebook of the chef from the south-west bedroom.\n The librarian will ask you a riddle - 3 is all you need to remember!\n In the north-west storage room you will find meat. Give it to the monster to tame it! NOW BEGONE TRAVELER!\n ");
400                 inventory.remove(command.getSecondWord());
401                 return;
402 
403             }
404         }
405 
406     }
407 
408     /**
409      * Pick up item command.
410      * Here the player can pick up items.
411      * Items are stored in the inventory.
412      */
413     private void take(Command command, ArrayList inventory)
414     {
415         if(!command.hasSecondWord())
416         {
417             // if there is no second word, we don't know where to go...
418             System.out.println("Take what? You need an item to take.Type inspect to see the items...\n");
419             return;
420         }
421         String secondWord=command.getSecondWord();
422         if( currentRoom.printItems().contains(secondWord))
423         {
424             int itemWeight = currentRoom.getItemWeight(secondWord);
425             if(itemWeight < storageSpace && (itemWeight + currentStorageSpace) < storageSpace)
426             {
427                 inventory.add(secondWord);
428                 currentStorageSpace+= itemWeight;
429                 spaceLeft=storageSpace-currentStorageSpace;
430                 System.out.println("Item added to the inventory. You have "+ spaceLeft+" kg of storage left!\n");
431             }
432 
433             else if((itemWeight + spaceLeft) > storageSpace && itemWeight < storageSpace )
434             {
435                 System.out.println("Inventory full! You have "+ spaceLeft+" kg of storage left!\n");
436 
437             }
438 
439             else
440             {
441                 System.out.println("Item too heavy to take. Your storage is 30kg. Type get + item + weight to check it!\n");
442 
443             }
```

```
444             if
445             (secondWord.equals("bookshelf"))
446             {
447                 System.out.println("You can not pick up this item!");
448 
449             }
450 
451         }
452     }
453 
454     /**
455      * Drop item command.
456      * Here the player can drop items.
457      * Items are removed from the inventory.
458      */
459     private void drop(Command command, ArrayList inventory)
460     {
461         if(!command.hasSecondWord())
462         {
463             // if there is no second word, we don't know where to go...
464             System.out.println("Remove what? You need an item to remove it.Type 'inspect inventory' to see the
    items...\n");
465             return;
466         }
467         String secondWord=command.getSecondWord();
468         if( inventory.contains(secondWord))
469         {
470             currentStorageSpace-=currentRoom.getItemWeight(secondWord);
471             inventory.remove(secondWord);
472             spaceLeft+=currentRoom.getItemWeight(secondWord);
473             System.out.println("Item removed from the inventory. You have "+ spaceLeft+" kg of storage left!\n");
474         }
475         else if (!inventory.contains(secondWord))
476         {
477             System.out.println("You do not have such item!");
478         }
479     }
480 
481     /**
482      * Get the items in the current room.
483      * Here we print out the items to the player.
484      *
485      */
486     private void getItems(Command command,ArrayList inventory)
487     {
488         if(!command.hasSecondWord())
489         {
490             if(currentRoom.printItems().charAt(currentRoom.printItems().length()-1)==' ')
491             {
492                 System.out.println("There are no items in this room!");
493             }
494             else
495                 System.out.println(currentRoom.printItems());
496 
497             return;
498         }
499         String secondWord=command.getSecondWord();
500         if(secondWord.toLowerCase().equals("inventory"))
501         {
502             if(inventory.size()==0)
503             {
504                 System.out.println("Your inventory is empty!\n ");
505             }
506             for(int i=0;i<inventory.size();i++)
507             {
508                 System.out.println("You are holding a/an " + inventory.get(i)+ " !\n");
509             }
510         }
511         else if (!secondWord.equals("inventory"))
512         {
513             System.out.println("Please enter a valid command!");
514         }
515     }
516 
517     /**
518      * Get the exits of the current room.
519      * Here we print out the exits to the player.
520      * If second and third command words are 'item'  and weight then get the weight of an item.
```

```
     */
    private void get(Command command)
    {
        if(!command.hasSecondWord())
        {
            // if there is no second word, we don't know where to go...
            System.out.println("Get what? Perhaps you mean 'exits'.../n");
            return;
        }
        String secondWord = command.getSecondWord();
        if(secondWord.toLowerCase().equals("exits"))
        {
            System.out.println(currentRoom.getLongDescription());
        }
        if(secondWord.toLowerCase().equals("items"))
        {
            System.out.println("Please type in a certain item and not 'item'!");
        }
        else if( currentRoom.printItems().contains(secondWord))
        {
            if(!command.hasThirdWord())
            {
                System.out.println("Get item's what? Perhaps you mean weight?\n");
                return;
            }

            String thirdWord=command.getThirdWord();
            if(thirdWord.equals("weight"))
            {
                System.out.println(currentRoom.getItemWeight(secondWord)+ "kg\n");
            }
        }

    }

    /**
     * Read the selected item aka the second command  word.
     * Here we print out the text which is read by the player.
     */
    private void read(Command command)
    {
        if(!command.hasSecondWord())
        {
            // if there is no second word, we don't know where to go...
            System.out.println("Read what? Specify an object...\n");
            return;
        }

        String object = command.getSecondWord();
        if(object.toLowerCase().equals("note"))
        {
            System.out.println("...to whoever reads this, I have failed to escape this place.\n I am writing this to warn YOU, traveler.\n If you wish to avoid my fate then you have to follow my instructions to the best of your ability.\n I do not know much about this place, no one does really. I have been given this knowledge by the person casted away here before my time and so\n I am passing my wisdom to you now.There is an exit from this place. To escape you need to find the key.\n Do not be fooled - for this task is nearly impossible. The key, once whole, \n is now shattered into 3 pieces, each and every one of them guarded by a vile and  mischievous creature somewhere in the halls of this mansion.\n Outsmarting them is the 'key' to getting your freedom back. Whenever unsure as to what to do remember to ask for 'help'. Good luck traveler...\n" );
        }
        else if (!object.equals("note"))
        {
            System.out.println("Please enter a valid command!");
        }
    }

    /**
     * Print out some help information.
     * Here we print a table with all the command decriptions to aid the player in their quest.
     */
    private void printHelp()
    {
        System.out.println("================================================================================");
        System.out.println("Your command words are: \n");
        parser.showCommands();
        System.out.println("================================================================================");
        System.out.println("--------------------------------------------------------------------------------");
        System.out.println("give item character - Gives a character some item.");
```

```
592         System.out.println("-----------------------------------------------------------------------------------");
593         System.out.println("take item - Adds item to the inventory.");
594         System.out.println("-----------------------------------------------------------------------------------");
595         System.out.println("drop item - Removes item from the inventory.");
596         System.out.println("-----------------------------------------------------------------------------------");
597         System.out.println("get exits - Prints out all the exits from the current room.");
598         System.out.println("-----------------------------------------------------------------------------------");
599         System.out.println("get item weight - Gets the weight of a certain item.");
600         System.out.println("-----------------------------------------------------------------------------------");
601         System.out.println("help - Prints out all the commands.");
602         System.out.println("-----------------------------------------------------------------------------------");
603         System.out.println("go exit - Moves the player to another room.");
604         System.out.println("-----------------------------------------------------------------------------------");
605         System.out.println("reply number - Used to answer to the Librarians's question.");
606         System.out.println("-----------------------------------------------------------------------------------");
607         System.out.println("back- Return to previous room.");
608         System.out.println("-----------------------------------------------------------------------------------");
609         System.out.println("quit - Quit game.");
610         System.out.println("-----------------------------------------------------------------------------------");
611         System.out.println("inspect- Inspects the current room for items.");
612         System.out.println("-----------------------------------------------------------------------------------");
613         System.out.println("inspect inventory- Returns the items the player is holding.");
614         System.out.println("-----------------------------------------------------------------------------------");
615         System.out.println("print location- Prints the current location of the player.");
616         System.out.println("-----------------------------------------------------------------------------------");
617         System.out.println("kill winnie- Takes winnie's life.");
618         System.out.println("-----------------------------------------------------------------------------------");
619         System.out.println("read note- Reads starting note.");
620         System.out.println("-----------------------------------------------------------------------------------");
621 
622     }
623 
624     /**
625      * Try to go in one direction. If there is an exit, enter the new
626      * room, otherwise print an error message. If you enter the transporter you get transported to a random room.
627      */
628     private void goRoom(Command command ,ArrayList roomsToBeRandomlyChosen)
629     {
630         if(!command.hasSecondWord())
631         {
632             // if there is no second word, we don't know where to go...
633             System.out.println("Go where?\n");
634             return;
635         }
636 
637         String direction = command.getSecondWord();
638         // Try to leave current room.
639 
640         visitedRooms.add(currentRoom);
641         Room nextRoom = currentRoom.getExit(direction);
642         //create a random index for Good winnie
643         Random rand=new Random();
644         int randomNumber = rand.nextInt(roomsToBeRandomlyChosen.size()-1);
645         //create a random index for Bad winnie
646         Random rand2=new Random();
647         int randomNumber2 = rand2.nextInt(roomsToBeRandomlyChosen.size()-1);
648 
649         Room nextNpcRoom=(Room)roomsToBeRandomlyChosen.get(randomNumber);
650         nextNpcRoom.getExit(getRandomNeighboringRoom()); //rooms which the winnie useless npc visits
651 
652         Room nextNpcRoomForBadNpc=(Room)roomsToBeRandomlyChosen.get(randomNumber2);
653         nextNpcRoomForBadNpc.getExit(getRandomNeighboringRoom()); //rooms which the bad winnie  npc visits
654         if (nextRoom == null)
655         {
656             System.out.println("There is no door!\n");
657             return;
658 
659         }
660         else if(nextRoom==library && hasKey1==true)
661         {
662             currentRoom = nextRoom;
663             System.out.println("This fragment has already been collected!"); //fragment already taken
664         }
665         else if(nextRoom==dungeonGrounds && hasKey3==true)
666         {
667             currentRoom = nextRoom;
668             System.out.println("This fragment has already been collected!");//fragment already taken
669         }
```

```
        else if(nextRoom==diningRoom && hasKey2==true)
        {
            currentRoom = nextRoom;
            System.out.println("This fragment has already been collected!");//fragment already taken
        }

        else if(nextRoom == transporter) // ransporter room
        {

            randomNumber = rand.nextInt(roomsToBeRandomlyChosen.size()-1);
            currentRoom = (Room)roomsToBeRandomlyChosen.get(randomNumber);
            System.out.println(" You are suddenly being transported!\n"+ currentRoom.getLongDescription());
            return;

        }
        else {
            if(nextRoom==nextNpcRoom)
            {
                currentRoom = nextRoom;
                if(winnie.getAlive()==true)
                {
                    System.out.println("\nA good winnie appears!.\n");
                    System.out.println(winnie.getDescription());
                }
            }
            if(nextRoom==nextNpcRoomForBadNpc)
            {
                currentRoom = nextRoom;
                if(winnie.getAlive()==false)
                {
                    System.out.println("\n You met the bad winnie again. Winnie is dead! You are a part of the
    dark side now!\n");
                }
                else
                {

                    System.out.println("\nA bad winnie appears!.\n");
                    System.out.println(badWinnie.getDescription());

                }
            }
            currentRoom = nextRoom;
            System.out.println(currentRoom.getLongDescription());
        }

    }

    /**
     * Go to the last room you visited.
     *
     */
    private void goBackRoom(int backStepCounter)
    {
        if(backStepCounter>visitedRooms.size()) //if you go back too many times stop!
        {
            System.out.println("You can't go back any further!");
            return;
        }
        if(visitedRooms.contains(visitedRooms.get(visitedRooms.size()-backStepCounter)))
        {
            currentRoom=visitedRooms.get(visitedRooms.size()-backStepCounter);
            System.out.println(currentRoom.getLongDescription());

        }
    }

    /**
     * "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;
        }
```

```
747         else
748         {
749             return true;  // signal that we want to quit
750         }
751     }
752 
753     /**
754      * Gets the exits of the current room the player is in and returns one of them randomly.
755      */
756     private String getRandomNeighboringRoom()
757     {
758         String exitString=currentRoom.returnExitString();
759         ArrayList<String> exits = new ArrayList<String>(Arrays.asList(exitString.split(" ")));
760         exits.add(exitString);
761         Random rand=new Random();
762         int randomNumber = rand.nextInt(exits.size()-1);
763         String randomExit = exits.get(randomNumber);
764         return randomExit;
765 
766     }
767 }
768 
```

```
/**
 * This class is part of the "Mansion escape" game.
 * "Mansion escape" is a very 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(for example - give meat monster).
 *
 * 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 and David J. Barnes and Mihail Bratanov
 * @version 1.0
 */

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.
     */
    public String getSecondWord()
    {
        return secondWord;
    }
      /**
     * @return The third word of this 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);
    }
}
```

```
/**
 * This class is part of the "Mansion escape" game.
 * "Mansion escape" is a very 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 and Mihail Bratanov
 * @version 1.0
 */

public class CommandWords
{
    // a constant array that holds all valid command words
    private static final String[] validCommands = {
        "go", "quit", "help","back" ,"read", "get", "inspect","take","drop", "give", "reply","print","kill"
    };

    /**
     * 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();
    }
}

```

```
import java.util.Set;
import java.util.HashMap;
import java.util.*;

/**
 * Class Room - a room in an adventure game.
 *
 * This class is part of the "Mansion escape" game.
 * "Mansion escape" is a very 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.
 *
 * @author  Michael Kölling and David J. Barnes and Mihail Bratanov
 * @version 1.0
 */

public class Room
{
    private String description;
    private HashMap<String, Room> exits;
    // stores exits of this room.
    private HashMap<String,Integer>items;


    /**
     * 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;
        exits = new HashMap<>();
        items=new HashMap<>();



    }

    /**
     * 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);
    }
    public void setItem(String item,int weight){
        items.put(item,weight);
    }

    /**
     * @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
     * @return A long description of this room
     */
    public String getLongDescription()
    {
        return "You are " + description + ".\n" + getExitString();
    }
    public String returnExitString(){
        return getExitString();
    }
    /**
```

```
79       * Return a description of the items in the room in the form:
80       *
81       *     Items: apple, book..
82       * @return A long description of this room
83       */
84      public String printItems()
85      {
86          return "The items in this room are:  " + getItemString();
87      }
88      /**
89       * Return a string describing the room's exits, for example
90       * "Exits: north west".
91       * @return Details of the room's exits.
92       */
93      private String getExitString()
94      {
95          String returnString = "Exits:";
96          Set<String> keys = exits.keySet();
97
98          for(String exit : keys) {
99              returnString += " " + exit;
100         }
101         if(returnString.equals(" ")){
102              returnString="?????";
103              return returnString;
104         }
105         return returnString;
106     }
107      /**
108      * Return a string describing the room's items, for example
109      * "Items: apple book".
110      * @return Details of the room's exits.
111      */
112     private String getItemString()
113     {
114         String returnString = " ";
115         Set<String> keys = items.keySet();
116         for(String item : keys) {
117             returnString += " " + item;
118         }
119         return returnString;
120     }
121
122    public int getItemWeight(String item){
123        return items.get(item);
124     }
125     /**
126      * Return the room that is reached if we go from this room in direction
127      * "direction". If there is no room in that direction, return null.
128      * @param direction The exit's direction.
129      * @return The room in the given direction.
130      */
131     public Room getExit(String direction)
132     {
133         return exits.get(direction);
134     }
135
136 }
137
138
```

```
import java.util.Scanner;

/**
 * This class is part of the "Mansion escape" game.
 * "Mansion escape" is a very 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 two 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 and Mihail Bratanov
 * @version 1.0
 */
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 three 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();
                    // 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();
    }
}

```

```
Project: zuul-better
Authors: Michael Kölling and David J. Barnes

This project is part of the material for the book

   Objects First with Java - A Practical Introduction using BlueJ
   Sixth edition
   David J. Barnes and Michael Kölling
   Pearson Education, 2016

This project is a simple framework for an adventure game. In this version,
it has a few rooms and the ability for a player to walk between these rooms.
That's all.

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

This project was written as the starting point of a small Java project.

The goal is to extend the game:

  - add items to rooms (items may have weight)
  - add multiple players
  - add commands (pick, drop, examine, read, ...)
  - (anything you can think of, really...)

Read chapter 8 of the book to get a detailed description of the project.

```