Michael Seiranian, K20127931

# The Prison Break

## PPA Assignment 2

### Game Description

The Prison Break is a text-based adventure game and is an evolved version of the World of Zuul created by Michael Kölling and David J. Barnes. The game begins by displaying text on the terminal, stating a welcome message and key information about the player's whereabouts, i.e., the items in the room they are in, the characters that are in the room, the items the player is holding, and other useful information for the player. Then the user is prompted to type a command, and using the commands of the game, the player must find a way to escape from prison. Important implementation features are HashMaps, Sets and ArrayLists. HashMaps were mainly used because a key piece of information about an object can be easily retrieved by calling on their value. Sets were used because of the fact that there can not be duplicates of the same object, and ArrayLists were used for the contrary, as in some cases, duplicated were desired and ArrayLists maintain order.

### Base Tasks and how they were completed

- **The game has several locations/rooms –**
  Using the Room class which already existed in the early version of the game, objects of type Room were created in the Game class, where the game is initiated from.
- **The player can walk through the locations –**
  Already implemented in the early version of the game, the rooms are linked to each other with exits, i.e., exiting one room enters another. Each Room object has its own HashMap called exits, with the keys being a String indicating direction (e.g. "north") and value the room that the direction leads to. The "go" command, followed by the direction, finds the next room and assigns it to current room.
- **There are items in some rooms. Every room can hold any number of items. Some items can be picked up by the player, others can't –**
  I created a class called "Item", which takes two parameters: name of item and its weight. Objects of type Item were created in the Game class under method "createItems" invoked in the constructor and HashMap "itemLocations" stored each item as the key, and their starting location as their value. I created a command "take", which the user can use to take an item, by typing take followed by the item name. If the item the user typed in is currently in the room the player is in, then the item is removed from the HashMap "itemLocations" and added to a Set "itemsHolding". However, if the item's weight is higher than the maximum weight the player can carry (defined by integer constant "maxWeight"), the item is not picked up and an appropriate error message is displayed.
- **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. –**
  When an item is picked up using the "take" command, the total weight of the items currently being carried is calculated using a for each loop iterating through each item and adding their weight to the totalWeight variable, which is compared to the maxWeight. If the item that the player wants to pick up causes the totalWeight to exceed maxWeight, the user gets an appropriate error message, only allowing them to carry items up to a certain total weight.
- **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 –**

When the command “use” is used followed by the “pliers” item, providing the player is at the exit and they are carrying the pliers and they have indicated they want to go north (which leads to outside the prison), the “useItem” method returns a true Boolean value to the processCommand method, which returns the boolean to the variable “finished” inside the main while loop. A victory and “good-bye” message are printed and the game ends.

- **Implement a command “back” that takes you back to the last room you’ve been in -**
  The room the player starts in is added to an ArrayList “roomHistory” and every time the player enters a new room, the new room is added to the ArrayList. Given the property of ArrayLists maintaining the order objects are stored in it, the back command places the player in the previous room in the ArrayList and removes the final room in the ArrayList. If the back command is used repeatedly, the ArrayList eventually reaches a length of one, consisting only of the room the player started in, and using the back command again simply provides the error message that the player is already in the room they started in.
- **Add at least four new commands –**
  Six new commands were added: "back", "take", "drop", "use", "talk to", "give". The implementations of “take” and “back” commands were already discussed. In the base task of the player being able to win, I briefly touched upon the “use” command. The command unlocks certain rooms with items that the player must be holding. Some of the items can unlock a room, and this information is stored in a HashMap “roomUnlocker”, with the item as the key and the room it unlocks as the value. The player must firstly indicate they want to go to a certain locked room using the “go” command, and then, providing they are carrying the item that unlocks the room, the player can use the “use” command and the room unlocks, and they are placed in that room.
  The command “drop” follows from the command “take”. If the item they want to drop is indeed in the player’s inventory (the “itemsHolding” set), the item is removed from the itemsHolding set and placed into the itemLocations HashMap, with the key as the item and the value as the current room the player is in.
  The “talk to” command is followed by the character the player wants to talk to. The command was completed by checking if the character is in the same room as the player and returning the default response of the character, which is predefined for each character in the second parameter when creating the character objects.
  The “give” command is followed by the item the player wants to give and then the character the player wants to give the item to. The command was completed by checking whether the item is in itemsHolding set and if the player’s room is the same as the character’s room. Since there is only one valid use for this command in the game, I stated in the code that if the item is “matches” and the character is “neighbour”, then initiate an event that leads to progression in the game, and in any other case, the player is displayed with appropriate error messages, as there is no other use for this command.

## Challenge tasks and how they were completed

- **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. –**
  Characters are initiated in the Game class, under method “createCharacters”, invoked inside the Game class constructor. Each created character is an object of type “Character”, a class I created for dealing with characters. Character starting locations are stored in HashMap “characterLocation”. To make characters move around by themselves, I created a method called “moveCharacters”, which is called every time the player changes rooms. The

characters only move to rooms that are positioned next to the room they are currently in, using the current room's "exits" HashMap.
To make the movement of the characters more independent, it takes a random number between 0 and 1, and if the number is 0, the character can move. This gives a 50% chance for the character to possibly move rooms. An array is created of the rooms neighbouring the current room, and a random number from 0 to array size - 1 picks the room the character should go to. However, there are exceptions when the character cannot move despite the random number being 0. For example, when the room the character has chosen to move to has no exits, the character would enter and get stuck. The game accounts for this potential error and prevents this from happening.

- **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 –**
  The commands "talk to" and "give" are three-word commands. To complete this, I set Command objects to take three string parameters instead of two (firstWord, secondWord and thirdWord). I also set the getCommand method in the Parser class to grab the first three words of the input instead of two and return three-word commands. Finally, in the Game class, under the commands, I set that if first word is "talk" and second word is "to", execute "talkToCharacter" method, and if command word is "give", execute "giveItem" method. Within the "give" method, the game checks whether the second word is an item the player is holding and if the third word is a character in current room.
- **Add a magic transporter room – every time you enter it you are transported to a random room in your game –**
  I created a room called "magicRoom" with no exits. Under the "go" command, when the game identifies that the room the player is entering is the magic room, the program creates an array of all unlocked rooms, and generates a random number to pick randomly from the array what the current room of the player should now be. An appropriate message is displayed, stating where the player has been teleported to.
- **Restrict access to certain rooms until an item is used –**
  I added a second parameter to Room objects of boolean value, which states if the room is locked. When the right item for unlocking the room is being carried and the player indicated that they want to go to a certain room which appears to be locked, the player can use the "use" command followed by the correct item, which changes the boolean value to false, and the room is unlocked. When the player uses the "go" command to go to a locked room, (i.e., nextRoom.getLocked() is true), they get an appropriate error message, stating they need an item to unlock the room.
- **Rewrite screen after every command –**
  After every event (e.g., a room change, an item pick up, an item drop, an error message, etc.), the game executes the "printWelcome" method, which prints the welcome message and key information on the terminal. The first line of the "printWelcome" method is "System.out.print('\u000C');", which clears the terminal before printing the key information, providing a less clogged and neater look to the game, and adding an element of challenge to the game, by removing the previous inputs and outputs, preventing the user from going back to check information.

## Code quality considerations

**Coupling –** The rooms, items and characters generated in the Game class were kept private, despite wanting to access information about them in their classes. For example, when for the moveCharacters method in Character class, I wanted to move all characters except the guard. Instead

of making the character objects public and static in Game class, I checked for the guard by checking if the description of the current character in the iteration is “guard”.

**Cohesion –** To avoid overcrowding the Game class, within many of the commands and the terminal printing method, there were method calls that would perform large tasks in one line. For example, in order to display the items that are in the player’s current room from a HashSet that has all items as the keys and their locations as their values, the program must iterate through the values of the HashSet, see which ones match the current room, create a set of the names of the items in that room and print out the set. To display the items in current room, all of these actions are grouped in a method called getItemsByRoom and called in one line in the printWelcome method in Game class.

**Responsibility-driven design –** When creating the game, the Game class was becoming too large. Looking back, I realised that many methods, sets, arrays and HashMaps all belong to other classes. After restructuring the code, the Game class has mostly only remained with methods that are directly linked with the terminal, initialising the game layout, creating objects in the layout, defining the commands or the player itself.

**Maintainability –** Consistent and comprehensive naming was used within the classes. For example, the Item and Character classes were very similar. To get a character’s and an item’s names, getCharacterDescription and getItemDescription methods were used. Character locations were stored in characterLocation HashMap and item locations were stored in itemLocation HashMap. In order to display the characters and items in current room, methods getItemsByRoom and getCharactersByRoom were used.

## Game walk through

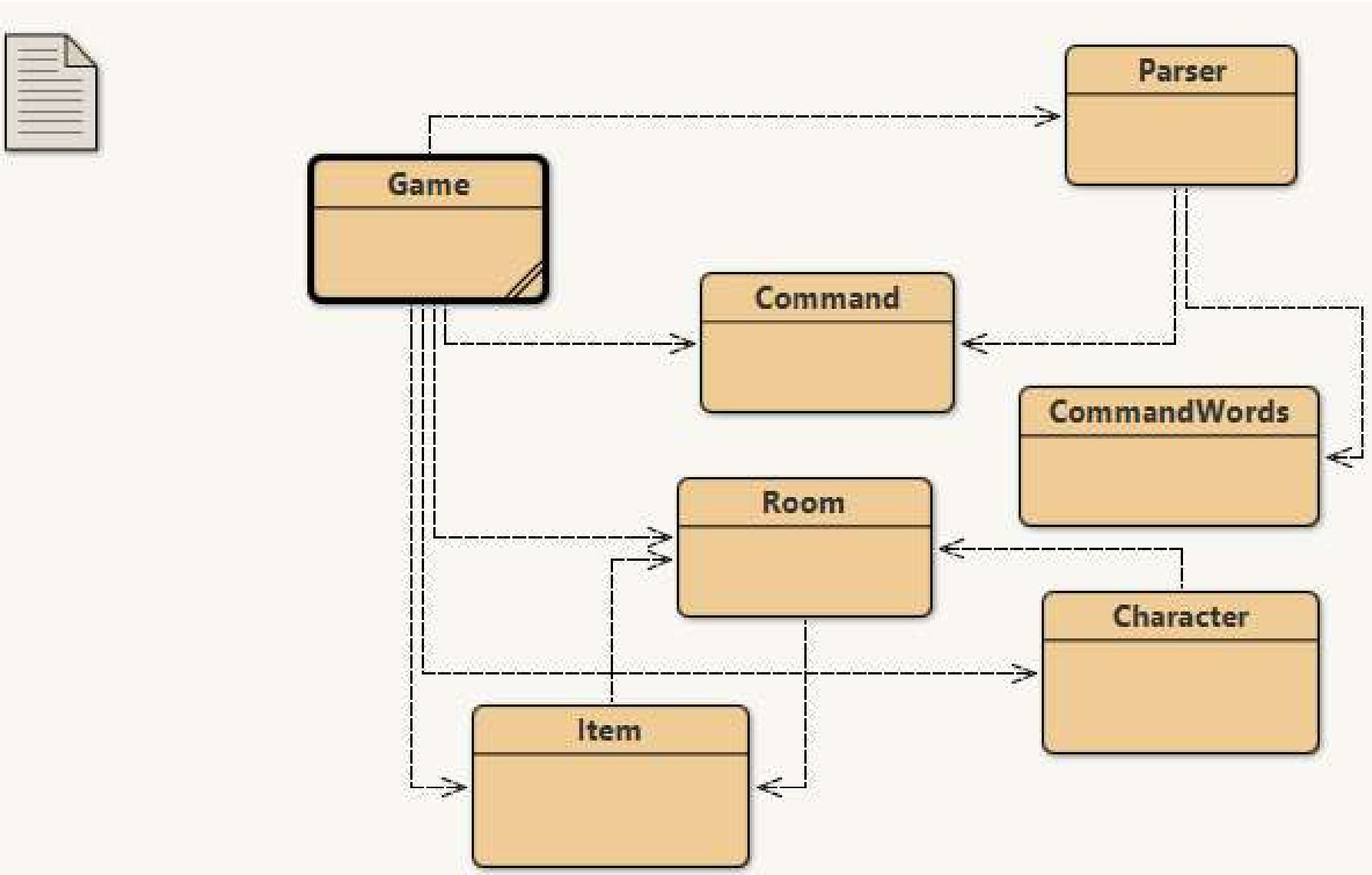
The player starts in their cell. To win, firstly, the player must do these things in no particular order:

- Go to staircase (firstly “go east” from cell, then “go north” from corridor), and take key, and go back to corridor and type “go south” from the corridor then type “use key” to unlock and enter guard’s room. They must take the “pliers” in this room, which will be needed in the final step of the game.
- Go to canteen (down from staircase), “take stool” and go to showers (north of staircase) and “drop stool”. The stool is needed in the showers to be able to reach the vents later when going up.
- Go to kitchen (west of canteen), “take matches”, locate neighbour (could be in any unlocked room as he moves around by himself), and type “give matches neighbour”. This starts a fire and guards are distracted from exit, and player can now escape.

Once these are done, from cell (room the player started in), “take wrench”. At this point, the player should be holding wrench and pliers, stool must be in showers and matches should have been given to neighbour. Go to showers and type “go up”, followed by “use wrench”. The player is now in the vents. Type “go north” twice, and finally “use pliers” to cut through the fence and win.

## Conclusion

The game has evolved to a good extent from the World of Zuul, with extra added functionality. However, it is not a great deal more exciting, as there are very little items and characters in the game. For this reason, when implementing the code, I considered creating it in a way that allows for easy item and character additions, with all functionality still being compatible. However, if more unlocked rooms were to be added in a certain arrangement, going into the magic room could teleport the player into a room that skips some locked ones, and new patching would be required.


Parser
Game
Command
CommandWords
Room
Character
Item

```java
import java.util.Set;
import java.util.Map;
import java.util.HashMap;
import java.util.HashSet;
/**
 * Class Item - an item in an adventure game.
 *
 * This class is part of the Prison Break application.
 * Prison Break is a very simple, text based adventure game.
 *
 * An "Item" represents an item in the scenery of the game.
 * It is in a room at all times. It can be picked up and dropped in different rooms by the main player
 * Some items can be used
 * Some items can be given to characters
 *
 * @author  Michael Seiranian
 * @version 2016.02.29
 */
public class Item
{
    private String itemDescription;
    private int weight;
    public static HashMap<Item, Room> itemLocation = new HashMap<Item, Room>();
//providing public access to static field to be accessed and edited by other classes

    /**
     * Constructor for objects of class Item
     *
     * @param itemDescription The name of the item
     * @param weight The weight of the item
     */
    public Item(String itemDescription, int weight)
    {
        this.itemDescription = itemDescription;
        this.weight = weight;
        itemLocation = new HashMap<>();
    }

    /**
     * Set item's room
     *
     * @param item The item we want to place somewhere
     * @param room The room we want to place the item in
     */
    public static void setItemRoom(Item item, Room room)
    {
        itemLocation.put(item, room);
    }

    /**
     * Remove item from room
     *
     * @param item The item we want to remove from the room
```

```
53      */
54     public static void RemoveItemFromRoom(Item item)
55     {
56         itemLocation.remove(item);
57     }
58 
59     /**
60      * Get string of all items in current room
61      * @param itemLocation The HashMap which has all items and their locations
62      * @param room The current room
63      * @return what getItemsString returns
64      */
65     public static String getItemsByRoom(HashMap<Item, Room> itemLocation, Room
room) {
66         Set<String> items = new HashSet<String>();
67         for (Map.Entry<Item, Room> entry : itemLocation.entrySet()) {
68             if (room == entry.getValue()) {
69                 items.add(entry.getKey().getItemDescription());
70             }
71         }
72 
73         return getItemsString(items);
74     }
75 
76     /**
77      * Continuation of getItemsByRoom method
78      * Converts Set of items to string and returns the string
79      * @param items The set of items passed on by getItemsByRoom
80      * @return String of all items in current room
81      */
82     public static String getItemsString(Set<String> items)
83     {
84         String returnString = "Items in this room:";
85         for(String item : items) {
86             returnString += " " + item;
87         }
88         return returnString;
89     }
90 
91     /**
92      * Get item's location
93      * @param item The item which we want to know the whereabouts of
94      * @return The room where the item is
95      */
96     public static Room getItemLocation(Item item)
97     {
98         return itemLocation.get(item);
99     }
100 
101     /**
102      * Get item's name
103      * @return String of item's name from constructor
104      */
105     public String getItemDescription()
106     {
```

```
107         return itemDescription;
108     }
109 
110     /**
111      * Get weight of item from constructor
112      * @return weight integer of item
113      */
114     public int getWeight()
115     {
116         return weight;
117     }
118 
119 }
120 
```

```
1  import java.util.*;
2  /**
3   *  This is the main class of the Prison Break game.
4   *
5   *  To play this game, create an instance of this class and call the "play"
6   *  method.
7   *
8   *  This main class creates and initialises all the others: it creates all
9   *  rooms, items, characters, creates the parser and starts the game.  It also
    evaluates and
10  *  executes the commands that the parser returns.
11  *
12  * @author  Michael Kölling, David J. Barnes and Michael Seiranian
13  * @version 2016.02.29
14  */
15
16 public class Game
17 {
18     private Parser parser;
19     private Room cell, corridor, guardoffice, staircase, showers, canteen,
    kitchen, vents, exit, outside, magicRoom, currentRoom, nextRoom;
20     private final int maxWeight = 10;        //maximum weight of objects the
    character can carry
21     private Set<Item> itemsHolding = new HashSet<Item>();   //stores items
    currently holding
22     private ArrayList<Room> roomHistory = new ArrayList<Room>();    //stores
    visited rooms
23     private Item key, stool, wrench, bed, matches, pliers, table;
24     private Character neighbour, guard, chef;
25     public HashMap<Character, Room> characterLocation =
    Character.characterLocation;
26     private boolean fire = false;
27     private boolean finished = false;
28     private boolean distractionText = true;
29
30     /**
31      * Create the game, initialise its internal map, items, players, their
    starting locations and which items unlock which rooms.
32      */
33     public Game()
34     {
35         createRooms();
36         addRoomsToSet();    //a set storing all rooms
37         initialiseRoomExits();      //sets the exits each room has
38         createItems();
39         setItemLocations();     //sets items' initial locations
40         setRoomUnlockers();     //sets which items unlock which rooms
41         createCharacters();
42         setCharacterLocations();    //sets items' initial locations
43         parser = new Parser();
44     }
45
46     /**
47      * Print out the main terminal for the player.
48      */
```

```
50      {
51          System.out.print('\u000C');
52          System.out.println("Welcome to the Prison Break!");
53          System.out.println("Prison Break is a new, boring adventure game.");
54          System.out.println("Type 'help' if you need help.");
55          System.out.println();
56          System.out.println("You are " + currentRoom.getShortDescription());
57          System.out.println(Item.getItemsByRoom(Item.itemLocation, currentRoom));
58          System.out.println(getItemsHoldingString());
59          System.out.println();

60  System.out.println(Character.getCharactersByRoom(Character.characterLocation,
    currentRoom));
61          System.out.println();
62          System.out.println("Total weight of items carrying:
    "+getTotalWeightOfItemsHolding());
63          System.out.println("Carrying weight limit: "+maxWeight);
64          System.out.println();
65          System.out.println(currentRoom.getExitString());
66          System.out.println();
67      }
68
69      /**
70       * Create all the rooms.
71       *
72       */
73      private void createRooms()
74      {
75          // create the rooms
76          cell = new Room("in your prison cell", false);
77          corridor = new Room("in the corridor", false);
78          guardoffice = new Room("in the guard's office", true);
79          staircase = new Room("on the staircase", false);
80          showers = new Room("in the shower room", false);
81          canteen = new Room("in the canteen", false);
82          kitchen = new Room("in the kitchen", false);
83          vents = new Room("in the vents", true);
84          exit = new Room("at the exit", true);
85          outside = new Room("outside", true);
86          magicRoom = new Room("in the magic room", false);
87
88          currentRoom = cell;  // start game in the cell
89          roomHistory.add(currentRoom);    //keeps track of visited rooms for the
    "back" command
90      }
91
92      /**
93       * Add rooms to set "allRooms".
94       *
95       */
96      private void addRoomsToSet() {
97          Room.allRooms.add(cell);
98          Room.allRooms.add(corridor);
99          Room.allRooms.add(guardoffice);
100         Room.allRooms.add(staircase);
```

```
101         Room.allRooms.add(showers);
102         Room.allRooms.add(canteen);
103         Room.allRooms.add(kitchen);
104         Room.allRooms.add(vents);
105         Room.allRooms.add(exit);
106     }
107 
108     /**
109      * Link room exits together.
110      *
111      */
112     private void initialiseRoomExits() {
113         // initialise room exits
114         cell.setExit("east", corridor);
115         corridor.setExit("west", cell);
116         corridor.setExit("south", guardoffice);
117         corridor.setExit("north", staircase);
118         guardoffice.setExit("north", corridor);
119         staircase.setExit("north", showers);
120         staircase.setExit("down", canteen);
121         staircase.setExit("south", corridor);
122         showers.setExit("south", staircase);
123         showers.setExit("up", vents);
124         canteen.setExit("up", staircase);
125         canteen.setExit("west", kitchen);
126         kitchen.setExit("east", canteen);
127         kitchen.setExit("north", magicRoom);
128         vents.setExit("down", showers);
129         vents.setExit("north", exit);
130         exit.setExit("south", vents);
131         exit.setExit("north", outside);
132     }
133 
134     /**
135      *  Create the items, sets their names and weights
136      *
137      */
138     private void createItems() {
139         // create the items
140         key = new Item("key", 4);
141         stool = new Item("stool", 4);
142         wrench = new Item("wrench", 4);
143         bed = new Item("bed", 200);
144         matches = new Item("matches", 1);
145         pliers = new Item("pliers", 4);
146         table = new Item("table", 100);
147     }
148 
149     /**
150      * Set the starting locations of the items.
151      *
152      */
153     private void setItemLocations() {
154         // initialise item locations
```

```
155         Item.setItemRoom(key, staircase);
156         Item.setItemRoom(stool, canteen);
157         Item.setItemRoom(wrench, cell);
158         Item.setItemRoom(bed, cell);
159         Item.setItemRoom(matches, kitchen);
160         Item.setItemRoom(pliers, guardoffice);
161         Item.setItemRoom(table, canteen);
162     }
163 
164     /**
165      * Set which item unlocks which room
166      */
167     private void setRoomUnlockers() {
168         // specify which items can unlock which rooms
169         Room.setRoomUnlocker(guardoffice, key);
170         Room.setRoomUnlocker(vents, wrench);
171         Room.setRoomUnlocker(outside, pliers);
172     }
173 
174     /**
175      * Create the characters. Set their name and default response
176      */
177     private void createCharacters() {
178         neighbour = new Character("neighbour", "Alright, I will help you. Just
promise me you will take care of my daughter when you're out.");
179         guard = new Character("guard", "I don't want to talk to you.");
180         chef = new Character("chef", "Leave me alone.");
181     }
182 
183     /**
184      * Set character starting locations
185      */
186     private void setCharacterLocations() {
187         Character.setCharacterRoom(neighbour, corridor);
188         Character.setCharacterRoom(guard, exit);
189         Character.setCharacterRoom(chef, kitchen);
190     }
191 
192     /**
193      *  Main play routine.  Loops until end of play.
194      */
195     public void play()
196     {
197         printWelcome();
198 
199         // Enter the main command loop.  Here we repeatedly read commands and
200         // execute them until the game is over.
201 
202         while (! finished) {
203             Command command = parser.getCommand();
204             finished = processCommand(command);
205         }
206 
207         System.out.println("Thank you for playing.  Good bye.");
```

```
208     }
209
210     /**
211      * Given a command, process (that is: execute) the command.
212      * @param command The command to be processed.
213      * @return true If the command ends the game, false otherwise.
214      */
215     private boolean processCommand(Command command)
216     {
217         boolean wantToQuit = false;
218
219         if(command.isUnknown()) {
220             printWelcome();
221             System.out.println("I don't know what you mean...");
222             return false;
223         }
224
225         String commandWord = command.getCommandWord();
226         if (commandWord.equals("help")) {
227             printHelp();
228         }
229         else if (commandWord.equals("go")) {
230             goRoom(command);
231         }
232         else if (commandWord.equals("take")) {
233             takeItem(command);
234         }
235         else if (commandWord.equals("drop")) {
236             dropItem(command);
237         }
238         else if (commandWord.equals("back")) {
239             back(command);
240         }
241         else if (commandWord.equals("use")) {
242             wantToQuit = useItem(command);
243         }
244         else if (commandWord.equals("talk")) {
245             if (command.getSecondWord().equals("to")) {
246                 talkToCharacter(command);
247             }
248         }
249         else if (commandWord.equals("give")) {
250             giveItem(command);
251         }
252         else if (commandWord.equals("quit")) {
253             wantToQuit = quit(command);
254         }
255         // else command not recognised.
256         return wantToQuit;
257     }
258
259     // implementations of user commands:
260
261     /**
```

```
262      * Print out some help information.
263      * Here we print some stupid, cryptic message and a list of the
264      * command words.
265      */
266     private void printHelp()
267     {
268         printWelcome();
269         System.out.println("You are trying to escape prison.");
270         System.out.println("You wander around in the prison.");
271         System.out.println();
272         System.out.println("Your command words are:");
273         parser.showCommands();
274     }
275 
276     /**
277      * Try to go in one direction. If there is an exit, enter the new
278      * room, otherwise print an error message.
279      * @param command The command to be processed.
280      */
281     private void goRoom(Command command)
282     {
283         if(!command.hasSecondWord()) {
284             printWelcome();
285             // if there is no second word, we don't know where to go...
286             System.out.println("Go where?");
287             return;
288         }
289 
290         String direction = command.getSecondWord();
291 
292         // Try to leave current room.
293         nextRoom = currentRoom.getExit(direction);
294 
295         if (nextRoom == null) {
296             printWelcome();
297             System.out.println("There is no door!");
298         }
299         else if (nextRoom.getLocked() &&
itemsHolding.contains(Room.getRoomUnlocker(nextRoom))) {
300             printWelcome();
301             System.out.println("Use the 'use' command, followed by the item name
to get " + nextRoom.getShortDescription());
302         }
303         else if (nextRoom.getLocked() && nextRoom == exit) {
304             printWelcome();
305             System.out.println("Can't go there! The guards are there, they must be
distracted and leave the prison exit");
306         }
307         else if (!nextRoom.getLocked() && nextRoom == outside) {
308             finished = true;
309             printWelcome();
310             System.out.println("Congratulations! You escaped!");
311         }
312         else if (nextRoom.getLocked()) {
```

```
313             printWelcome();
314             System.out.println("Can't get in! You need some item to get " +
nextRoom.getShortDescription());
315         }
316         else if (nextRoom == magicRoom) {
317             roomHistory.add(currentRoom);
318             Random random = new Random();
319             int randomNumber = random.nextInt(Room.getUnlockedRoomsSet().size());
320             currentRoom = Room.getUnlockedRoomsArray()[randomNumber];
321             printWelcome();
322             System.out.println("You entered the magic transporter room.");
323             System.out.println("You have been teleported and are now " +
currentRoom.getShortDescription()+"!");
324         }
325         else if (currentRoom == showers && nextRoom == vents &&
Item.getItemLocation(stool) != showers) {
326             printWelcome();
327             System.out.println("Too high! You need the stool in this room to be
able to climb into the vents.");
328         }
329         else {
330             currentRoom = nextRoom;
331             roomHistory.add(nextRoom);
332             Character.moveCharacters();
333             printWelcome();
334             if (fire==true) {
335                 System.out.println("Your neighbour has started a fire, the guards
are distracted, you can escape!");
336                 fire = false;
337             }
338         }
339     }
340 
341     /**
342      * Try to take item. If there such item in current room, take the item,
343      * otherwise print an error message.
344      * @param command The command to be processed.
345      */
346     private void takeItem(Command command)
347     {
348         if(!command.hasSecondWord()) {
349             printWelcome();
350             // if there is no second word, we don't know what to take...
351             System.out.println("Take what?");
352             return;
353         }
354 
355         String itemText = command.getSecondWord();
356         Item itemTaken = null;
357         for(Item item: Item.itemLocation.keySet()) {        //shows the items that
are in this room
358             if(item.getItemDescription().equals(itemText))      //finds the item
that matches user's input
359             {
360                 itemTaken = item;
```

```
361             }
362         }
363 
364         if (getItemsHoldingString().contains(itemText)) {
365             printWelcome();
366             System.out.println("You are already carrying this item.");
367             return;
368         }
369 
370         if (itemTaken == null) {
371             printWelcome();
372             System.out.println("There is no such item!");
373             return;
374         }
375 
376         if (Item.getItemLocation(itemTaken) != currentRoom) {
377             printWelcome();
378             System.out.println("There is no such item in this room!");
379         }
380         else if (itemTaken.getWeight()>maxWeight) {
381             printWelcome();
382             System.out.println("You can't carry this item. It is too heavy.");
383         }
384         else if (getTotalWeightOfItemsHolding() + itemTaken.getWeight() >
maxWeight) {
385             printWelcome();
386             System.out.println("Weight limit exceeded. Can't take item");
387         }else{
388             itemsHolding.add(itemTaken);
389             Item.RemoveItemFromRoom(itemTaken);
390             printWelcome();
391             System.out.println(itemTaken.getItemDescription() + " taken");
392         }
393     }
394 
395     /**
396      * @return string of items currently holding
397      */
398     private String getItemsHoldingString()
399     {
400         String returnString = "Items currently holding:";
401         for(Item item : itemsHolding) {
402             returnString += " " + item.getItemDescription();
403         }
404         return returnString;
405     }
406 
407     /**
408      * @return total weight of items holding
409      */
410     private int getTotalWeightOfItemsHolding()
411     {
412         int totalWeight=0;
413         for (Item item: itemsHolding) {
414             totalWeight += item.getWeight();
```

```
415         }
416         return totalWeight;
417     }
418 
419     /**
420      * Try to drop item. If you are holding such item, drop the item,
421      * otherwise print an error message.
422      * @param command The command to be processed.
423      */
424     private void dropItem(Command command)
425     {
426         if(!command.hasSecondWord()) {
427             printWelcome();
428             // if there is no second word, we don't know what to drop...
429             System.out.println("Drop what?");
430             return;
431         }
432 
433         String itemText = command.getSecondWord();
434 
435         Item itemDropping = null; //The item that we want to drop
436         for(Item item: itemsHolding) {
437             if(item.getItemDescription().equals(itemText))
438             {
439                 itemDropping = item;
440             }
441         }
442 
443         if (!itemsHolding.contains(itemDropping)) {
444             printWelcome();
445             System.out.println("There is no such item in your inventory!");
446         }
447         else{
448             itemsHolding.remove(itemDropping);
449             Item.setItemRoom(itemDropping, currentRoom);
450             printWelcome();
451             System.out.println(itemDropping.getItemDescription() + " dropped " +
currentRoom.getShortDescription());
452         }
453     }
454 
455     /**
456      * Use item. If the item exists in current items holding and it is the
457      * correct item the unlocks the room, unlock the room and go in it,
458      * otherwise print an error message.
459      * @param command The command to be processed.
460      * @return return true when game must be finished
461      */
462     private boolean useItem(Command command)
463     {
464         if(!command.hasSecondWord()) {
465             printWelcome();
466             // if there is no second word, we don't know what to drop...
467             System.out.println("Use what?");
```

```
468             return false;
469         }
470 
471         String itemText = command.getSecondWord();
472 
473         Item itemUsing = null; //The item that we wnat to drop
474         for(Item item: itemsHolding) {
475             if(item.getItemDescription().equals(itemText))
476             {
477                 itemUsing = item;
478             }
479         }
480 
481         if (!itemsHolding.contains(itemUsing)) {
482             printWelcome();
483             System.out.println("There is no such item in your inventory!");
484         }
485         else if (itemUsing != Room.getRoomUnlocker(nextRoom)) {
486             printWelcome();
487             System.out.println("You can not use this item here!");
488         }
489         else if (currentRoom == showers && nextRoom == vents &&
Item.getItemLocation(stool) != showers) {
490             printWelcome();
491             System.out.println("You need to have the stool in this room to reach
the vents.");
492         }
493         else{
494             itemsHolding.remove(itemUsing);
495             nextRoom.Unlock();
496             Room.removeRoomUnlocker(nextRoom);
497             currentRoom = nextRoom;
498             System.out.println(nextRoom.getShortDescription());
499             if (nextRoom==outside) {
500                 printWelcome();
501                 System.out.println("Congratulations! You escaped!");
502                 return true;
503             }
504             else {
505                 printWelcome();
506                 System.out.println("Room unlocked");
507             }
508         }
509         return false;
510     }
511 
512     /**
513      * Go back. Goes back a room.
514      * @param command The command to be processed.
515      */
516     private void back(Command command)
517     {
518         if(command.hasSecondWord()) {
519             printWelcome();
```

```
520             System.out.println("Back where?");
521         }
522         else {
523             if (roomHistory.size() == 1){     // make the previous room the current one
524                 currentRoom = roomHistory.get(0);
525                 printWelcome();
526                 System.out.println("You are in the room you started");
527             }
528             else {
529                 currentRoom = roomHistory.get(roomHistory.size()-2);
530                 roomHistory.remove(roomHistory.size()-1);
531                 printWelcome();
532             }
533         }
534     }
535 
536     /**
537      * Talk to character. If you are in a room with a character, you can talk to them,
538      * otherwise print an error message.
539      * @param command The command to be processed.
540      */
541     private void talkToCharacter(Command command) {
542         if(!command.hasThirdWord()) {
543             printWelcome();
544             // if there is no third word, we don't know who to talk to...
545             System.out.println("Talk to who?");
546             return;
547         }
548 
549         String characterText = command.getThirdWord();
550 
551         Character characterToTalkTo = null;
552         for(Character character: Character.characterLocation.keySet()) { //shows the characters that are in this room
553             if(character.getCharacterDescription().equals(characterText)) //finds the character that matches user's input
554             {
555                 characterToTalkTo = character;
556             }
557         }
558 
559         if (characterToTalkTo == null) {
560             printWelcome();
561             System.out.println("There is no such character!");
562             return;
563         }
564 
565         if (Character.getCharacterLocation(characterToTalkTo) != currentRoom) {
566             printWelcome();
567             System.out.println("There is no such character in this room!");
568         }
569         else if (characterToTalkTo == neighbour) {
570             printWelcome();
```

```
571             if (distractionText == true) {
572                 System.out.println("you: Hey man, can you cause a distraction while I escape prison?");
573             }
574             distractionText = false;
575             System.out.println(characterToTalkTo.getCharacterDescription()+": "+characterToTalkTo.getResponse());
576             characterToTalkTo.setResponse("Get me something I can start a fire with");
577         }
578         else{
579             printWelcome();
580             System.out.println(characterToTalkTo.getCharacterDescription()+": "+characterToTalkTo.getResponse());
581         }
582     }
583 
584     /**
585      * Give an item to a character. If the item you are giving is matches and the character is the neighbour,
586      * The neighbour starts a fire after you move again,
587      * otherwise print an appropriate error message.
588      * @param command The command to be processed.
589      */
590     private void giveItem(Command command) {
591         if(!command.hasSecondWord()) {
592             printWelcome();
593             // if there is no second word, we don't know what to give...
594             System.out.println("Give what?");
595             return;
596         }
597 
598         String itemText = command.getSecondWord();
599 
600         Item itemGiving = null; //The item that we want to give
601         for(Item item: itemsHolding) {
602             if(item.getItemDescription().equals(itemText))
603             {
604                 itemGiving = item;
605             }
606         }
607 
608         if (!itemsHolding.contains(itemGiving)) {
609             printWelcome();
610             System.out.println("There is no such item in your inventory!");
611         }
612         else {
613             if(!command.hasThirdWord()) {
614                 printWelcome();
615                 // if there is no third word, we don't know who to give the item to...
616                 System.out.println("Give item to who?");
617                 return;
618             }
619 
```

```
621
622             Character characterToGiveTo = null;
623             for(Character character: Character.characterLocation.keySet())
{         //shows the characters that are in this room

624 if(character.getCharacterDescription().equals(characterText))      //finds the
character that matches user's input
625                 {
626                     characterToGiveTo = character;
627                 }
628             }
629
630             if (characterToGiveTo == null) {
631                 printWelcome();
632                 System.out.println("There is no such character!");
633                 return;
634             }
635
636             if (Character.getCharacterLocation(characterToGiveTo) != currentRoom)
{
637                 printWelcome();
638                 System.out.println("There is no such character in this room!");
639             }
640             else if (characterToGiveTo == neighbour && itemGiving == matches) {
641                 itemsHolding.remove(itemGiving);
642                 fire = true;
643                 exit.Unlock();
644                 neighbour.setResponse("You can escape now, go!!");
645                 Character.setCharacterRoom(guard, kitchen);
646                 printWelcome();
647                 System.out.println(characterToGiveTo.getCharacterDescription() +
": Get moving and I'll start the fire to distract the guards. Look after my
daughter!");
648             }
649             else {
650                 printWelcome();
651                 System.out.println(characterToGiveTo.getCharacterDescription() +
": I don't know what to do with this.");
652             }
653         }
654     }
655
656     /**
657      * "Quit" was entered. Check the rest of the command to see
658      * whether we really quit the game.
659      * @param command The command to be processed.
660      * @return true, if this command quits the game, false otherwise.
661      */
662     private boolean quit(Command command)
663     {
664         if(command.hasSecondWord()) {
665             printWelcome();
666             System.out.println("Quit what?");
667             return false;
668         }
```

```
671             }
672         }
673
674 }
675
```

```java
import java.util.Set;
import java.util.Map;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Random;
/**
 * Class Character - a character in an adventure game.
 *
 * This class is part of the Prison Break application.
 * Prison Break is a very simple, text based adventure game.
 *
 * A "Character" represents a secondary character (not the main character) in the game.
 * It is in a room at all times. Each character has a name and response to the main player.
 *
 * @author  Michael Seiranian
 * @version 2016.02.29
 */
public class Character
{
    private String characterDescription;
    private String response;
    public static HashMap<Character, Room> characterLocation;

    /**
     * Constructor for objects of class Character
     * @param characterDescription The name of the character
     * @param response The default response character gives to main player
     */
    public Character(String characterDescription, String response)
    {
        this.characterDescription = characterDescription;
        this.response = response;
        characterLocation = new HashMap<>();
    }

    /**
     * Place characters in rooms
     * @param character The character which we want to place in a room
     * @param room The room where we want to place the character
     */
    public static void setCharacterRoom(Character character, Room room)
    {
        characterLocation.put(character, room);
    }

    /**
     * Get string of all characters in current room
     * @param characterLocation The HashMap which has all characters and their locations
     * @param room The current room
     * @return what getCharactersString returns
     */
```

```
52      public static String getCharactersByRoom(HashMap<Character, Room>
    characterLocation, Room room) {
53          Set<String> characters = new HashSet<String>();
54          for (Map.Entry<Character, Room> entry : characterLocation.entrySet()) {
55              if (room == entry.getValue()) {
56                  characters.add(entry.getKey().getCharacterDescription());
57              }
58          }
59
60          return getCharactersString(characters);
61      }
62
63      /**
64       * Continuation of getCharactersByRoom method
65       * Converts Set of characters to string and returns the string
66       * @param characters The set of characters passed on by getCharactersByRoom
67       * @return String of all characters in current room
68       */
69      public static String getCharactersString(Set<String> characters)
70      {
71          String returnString = "Characters in this room:";
72          for(String character : characters) {
73              returnString += " " + character;
74          }
75          return returnString;
76      }
77
78      /**
79       * Get character location
80       * @param character The character that we want to know the location of
81       * @return the Room of the character
82       */
83      public static Room getCharacterLocation(Character character)
84      {
85          return characterLocation.get(character);
86      }
87
88      /**
89       * Get character's response
90       * @return the response of the character
91       */
92      public String getResponse()
93      {
94          return response;
95      }
96
97      /**
98       * Change character's response
99       * @param newResponse The new response of the character
100      */
101     public void setResponse(String newResponse)
102     {
103         response = newResponse;
104     }
```

```
105
106     /**
107      * Get character description
108      * @return name of character
109      */
110     public String getCharacterDescription()
111     {
112         return characterDescription;
113     }
114
115     /**
116      * Move characters around if the character is not guard
117      */
118     public static void moveCharacters()
119     {
120         for (Character character : characterLocation.keySet()) {
121             if (character.getCharacterDescription().equals("guard") == false) {
122                 Random random = new Random();
123                 int randomNumber = random.nextInt(2);
124                 if (randomNumber == 0) {    //giving it a 50% chance for the
character to move, to make their movement more random and independent
125                     Room currentRoom = getCharacterLocation(character);     //find
where character is
126                     Random random1 = new Random();
127                     Object[] roomsObj =
currentRoom.exits.values().toArray();       //get its locations exits and convert
to array
128                     Room[] rooms = new Room[roomsObj.length];
129                     System.arraycopy(roomsObj, 0, rooms, 0,
roomsObj.length);        //copy object array into Room array
130                     int randNum = random1.nextInt(rooms.length);
131                     Room nextRoom = rooms[randNum];     //pick random room from
the exits
132                     Object[] nextRoomsObj = nextRoom.exits.values().toArray();
//get nextRoom's exits
133                     if (!nextRoom.getLocked() && nextRoomsObj.length>0) {
//if nextRoom is not locked and nextRoom has more than 0 exits, the character can
proceed with moving to the next room
134                         characterLocation.put(character, nextRoom);
135                     }
136                 }
137             }
138         }
139     }
140 }
141
```

```java
/**
 * This class is part of the "Prison Break" application.
 *
 * 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 and a third word
 *
 * 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 Michael Seiranian
 * @version 2016.02.29
 */

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.
     */
    public Command(String firstWord, String secondWord, String thirdWord)
    {
        commandWord = firstWord;
        this.secondWord = secondWord;
        this.thirdWord = thirdWord;
    }

    /**
     * Return the command word (the first word) of this command. If the
     * command was not understood, the result is null.
     * @return The command word.
     */
    public String getCommandWord()
    {
        return commandWord;
    }

    /**
     * @return The second word of this command. Returns null if there was no
     * second word.
     */
    public String getSecondWord()
    {
        return secondWord;
```

```
    }

    /**
     * @return the third word of the command. Returns null if there was no
     * third word.
     */
    public String getThirdWord()
    {
        return thirdWord;
    }

    /**
     * @return true if this command was not understood.
     */
    public boolean isUnknown()
    {
        return (commandWord == null);
    }

    /**
     * @return true if the command has a second word.
     */
    public boolean hasSecondWord()
    {
        return (secondWord != null);
    }

    /**
     * @return true if the command has a third word.
     */
    public boolean hasThirdWord()
    {
        return (thirdWord != null);
    }
}


```

```java
/**
 * This class is part of the "Prison Break" application.
 *
 * 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 Michael Seiranian
 * @version 2016.02.29
 */

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

    /**
     * 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();
    }
}

```

```java
import java.util.Set;
import java.util.HashMap;
import java.util.HashSet;
/**
 * Class Room - a room in an adventure game.
 *
 * This class is part of the Prison Break application.
 * Prison Break 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, David J. Barnes and Michael Seiranian
 * @version 2016.02.29
 */

public class Room
{
    private String description;
    private boolean locked;
    public HashMap<String, Room> exits;        // stores exits of this room.
    public static HashMap<Room, Item> roomUnlocker; // stores item that unlocks the room
    public static Set<Room> allRooms = new HashSet<Room>();
    private static Set<Room> unlockedRooms = new HashSet<Room>();

    /**
     * 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.
     * @param locked If the room is locked or not.
     */
    public Room(String description, boolean locked)
    {
        this.description = description;
        this.locked = locked;
        exits = new HashMap<>();
        roomUnlocker = new HashMap<>();
    }

    /**
     * Define an exit from this room.
     * @param direction The direction of the exit.
     * @param neighbour  The room to which the exit leads.
     */
    public void setExit(String direction, Room neighbour)
    {
        exits.put(direction, neighbour);
    }

    /**
     * @return The short description of the room
```

```
     * (the one that was defined in the constructor).
     */
    public String getShortDescription()
    {
        return description;
    }

    /**
     * @return true if room is locked
     */
    public boolean getLocked()
    {
        return locked;
    }

    /**
     * Unlocks room
     */
    public void Unlock()
    {
        locked = false;
    }

    /**
     * Return a string describing the room's exits, for example
     * "Exits: north west".
     * @return Details of the room's exits.
     */
    public String getExitString()
    {
        String returnString = "Exits:";
        Set<String> keys = exits.keySet();
        for(String exit : keys) {
            returnString += " " + exit;
        }
        return returnString;
    }

    /**
     * Return the room that is reached if we go from this room in direction
     * "direction". If there is no room in that direction, return null.
     * @param direction The exit's direction.
     * @return The room in the given direction.
     */
    public Room getExit(String direction)
    {
        return exits.get(direction);
    }

    /**
     * Get the item that unlocks the room
     * @param room The room that we want to know the unlocker of
     * @return the item that unlocks the room
     */
```

```
    public static Item getRoomUnlocker(Room room)
    {
        return roomUnlocker.get(room);
    }

    /**
     * Set the item that unlocks the room
     * @param room The room that the item unlocks
     * @param item The item that unlocks the room
     */
    public static void setRoomUnlocker(Room room, Item item)
    {
        roomUnlocker.put(room, item);
    }

    /**
     * Remove room and item pair since the unlock has occured
     * @param room The room we want to remove
     */
    public static void removeRoomUnlocker(Room room)
    {
        roomUnlocker.remove(room);
    }

    /**
     * Get set of all unlocked rooms
     * @return unlockedRooms The Set that contains all unlocked rooms
     */
    public static Set<Room> getUnlockedRoomsSet()
    {
        for (Room room: allRooms) {
            if (room.getLocked() == false) {
                unlockedRooms.add(room);
            }
        }
        return unlockedRooms;
    }

    /**
     * Get array of all unlocked rooms
     * @return unlockedRoomsArray The Array that contains all unlocked rooms
     */
    public static Room[] getUnlockedRoomsArray()
    {
        Room[] unlockedRoomsArray = getUnlockedRoomsSet().toArray(new
Room[getUnlockedRoomsSet().size()]);
        return unlockedRoomsArray;
    }
}


```

```java
import java.util.Scanner;

/**
 * This class is part of the "Prison Break" application.
 *
 * This parser reads user input and tries to interpret it as an "Adventure"
 * command. Every time it is called it reads a line from the terminal and
 * tries to interpret the line as a three word command. It returns the command
 * as an object of class Command.
 *
 * The parser has a set of known command words. It checks user input against
 * the known commands, and if the input is not one of the known commands, it
 * returns a command object that is marked as an unknown command.
 *
 * @author  Michael Kölling, David J. Barnes and Michael Seiranian
 * @version 2016.02.29
 */
public class Parser
{
    private CommandWords commands;  // holds all valid command words
    private Scanner reader;         // source of command input

    /**
     * Create a parser to read from the terminal window.
     */
    public Parser()
    {
        commands = new CommandWords();
        reader = new Scanner(System.in);
    }

    /**
     * @return The next command from the user.
     */
    public Command getCommand()
    {
        String inputLine;   // will hold the full input line
        String word1 = null;
        String word2 = null;
        String word3 = null;

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

        inputLine = reader.nextLine();

        // Find up to 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 if(commands.isCommand(word1+" "+word2)) {
            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.

```