Bond's Adventures

Description of the game:

User Level Description:
The game is basically an adventure text-based game, based on the framework of 'The Word of Zuul Adventure'. The player embodies James Bond, a high level agent which is appointed to complete a high-risk mission for the sake of the CIA. The mission consist of finding and killing a well known murderer in a house. Thus, the map of the game consists of common types of rooms that you can find in a house. In total, there are 10 rooms and 3 floors. Kryzko, the serial-killer, is sleeping in a locked room, while his guard is moving around the house to watch-out for any intruder. There are a total of 8 items spread among all rooms. They can vary from common objects (i.e tv, paint, keys, knife), to weapons (i.e gun, ammo, poison). James Bond is carrying with him a backpack that serves as the player's inventory. However, he can carry with him up to a certain limited amount of weight (in the game it is set to 1600 grams), to be the lightest and the fastest possible. Therefore, the player should be careful on the usefulness of the items he takes. At the beginning of the game, James Bond has no weapon on him, and thus, the player should find weapons in the house before being found by the guard, otherwise the game is lost. As it is a text-based game, the game is played through commands entered by the player. The game recognizes a list of well defined commands that can be shown using the 'help' command.

Implementation features:
In total there are 11 classes composing the game. First I created the Player class which holds all the characteristics of the player. That is to say, its current room, its previous room, its inventory and a boolean isKilled to know whether or not the player is still alive. The classes Room, Item, Character, CharacterManger and Inventory are responsible for the game structure. The classes Parser, Command and CommandWords are responsible for reading an input from the player and interpret it as an command. Then there is the Game class, which initializes the game structure and executes all commands received, and the Launcher class which runs the game when executed. Finally we have the SoundEffect class which is at the origin of all sounds heard in the game. I implemented three sounds: one background sound that keeps playing while the game is on, an opening door sound effect that appears every time the player changes room and finally a scary deep voice that appears when Kryzko is found.

Base Task completed and How ?

- <u>The game has several locations/rooms</u>
  I created a Room Class that holds all the characteristics of a room, that is to say its: name, description, items, exits and key. It also holds a boolean that shows whether or not the room is locked. All rooms are instantiated in the Game Class and stored in a HashMap called roomHashMap with the string name of the room as the key and the room object as the value.

- <u>The player can walk through the locations.</u>

Each room has one or more exits, which are instantiated in the Game Class and stored in a HashMap in the Room Class with directions as its keys and exit rooms as its values. There are 6 possible directions: “west”, “east”, “south”, “north”, “down” and “up”. To move from a room to another, a go command has been implemented. It reads the direction written and puts the player in the wanted room.

- There are items in some rooms. Some items can be picked up by the player, others can’t.
  I first created an Item Class that holds the name, the description and the weight of each item object. I instantiated each item in the Game Class and then spread the items among all the rooms. To do so, I created for each room a HashMap containing all items of the room, with the name of the items as its keys and the items objects as its values. A boolean canBeCarried stores whether or not the item can be carried.

- The player can carry some items with him. The player can carry items up to a certain total weight.
  I created an Inventory Class. Each Inventory object holds a HashMap of the name of the items contained as keys and the items objects as values. It also holds a maximum weight which can be carried in the inventory. The player inventory is set in the Game Class at 1600 grams. In the inventory class I implemented a method that returns a boolean that tells whether or not an item can be added to the inventory by comparing the max weight and the inventory weight + the weight of the item wanted to grab. It is used in the grab method inside the Game Class.

- The player can win
  In the main play loop in the Game Class. I added an if statement that checks after each command if kryzko is dead, if yes the boolean is true and the player wins !

- Implement a command “back” that takes you back to the last room you’ve been in.
  To implement the back command I added a field, previousRoom, to the Player class that stores the room that the player was in previously, retrieving this information from the “go room” method in the Game Class. The “back” method sets the current room of the player to its previous room. And if the player has no previous room he gets noticed.

- Add at least four new commands
  In total I added 9 new commands. What I did is that, for each new command I wanted to add, I added the command word and its description in the commandWords HashMap inside the CommandWords Class. Then inside the Game Class I implemented a new method linked to the newly created command Word by the ‘processCommand’ method so that each time the common word is typed the command the program knows which command to execute. All new commands can be found by typing the help command.

Challenge Tasks completed and How ?

- Add characters to your game. They can move around by themselves.
  For this task, I created a new class named Character. Each object of this class holds a name, a description, an inventory, a currentRoom and a boolean isKilled. The class Character is very similar to the Player class, in that it has approximately the same fields and methods. However, characters are not controlled by the player and some of them move around by themselves, thanks to a method named ‘moveRandomly’.

  This method is implemented inside another Class : CharacterManager that holds a HashMap which combines each character with a room. Basically, what I did is that I’ve created a ‘generateRandomExit’ method inside my Room class that I use in my ‘moveRandomly’ method. This method allows me to

get a random exit among the exits of the character's current room, then I remove the character from its current room and I add it into the new randomly generated room. The 'moveRandomly' method is called when the player changes rooms.

- Extend the praser to recognize three-word commands.
  I just used the initial framework of the Parser Class and added a third field 'thirdword' in the Command Class.

- Add a magic transporter room
  To do so, I added a HashMap field to the Game Class so as to store all unlocked rooms of the game. Then, I created a method to generate a random room and a 'teleportPlayer' method that teleports the player to the random room generated.

Additional challenge tasks :

- Having the possibility to begin the game with locked rooms that can be opened using unique keys. Each room has its own keys. I also implemented a new 'open' command to open the room if the inventory of the player contains the corresponding key.
- Implemented background sound & sound effects by importing SoundEffect Class from the internet.

Code Quality Example :

Coupling
I considered coupling when I created my characterManager Class. What happened is that I wanted my Character Class to have a current room field so that I can access any character's current room easily. However, when I was doing the method that moves characters randomly, I realized that I also needed to add a field named "characterIn Room" in my Room Class. As a consequence, my Room and Character classes were really tightly coupled. So I decided to create a new class that I called characterManager which has as a field a HashMap containing rooms as its keys and Characters as its values. The HashMap stores all characters' positions in the map. Thanks to this Class I no longer encountered any difficulties in trying to make my 'moveRandomly' method and I have reduced to nothing the interdependence of both my Room and Character classes.

Cohesion
I considered cohesion when I wanted to create an inventory for my player. At first, what I did is that I created an object of the Room class that played the role of an inventory. The unique difference that it had with the others same class objects, it's that it was not holding any exits but rather only items. But then, I had to implement all the methods that were useful only for the inventory object and were not used at all by all the other 'true' room objects of the class. At this point, I thought that everything would get messy, so I decided to create a new Inventory class especially reserved for the characteristics of the inventory and all its methods. Hence, increasing cohesion and lowering confusion of my Room Class.

Responsibility-driven design
I considered RDD for my 'teleportPlayer' method for the transporter room. At first, I created the method inside the Player Class, thinking that it concerned the player solely. However, when I wanted to get the randomly generated room I couldn't because the data was in the Game Class. Hence, I decided to put the 'teleportPlayer' method in the Game Class, where the data is easily accessible.

Maintainability
I considered maintainability in the character class. In fact, it is quite easy to extend this class with new functionalities. At first, characters had not the boolean isKilled, I added it afterwards and it didn't impact badly

any part of the code. Also, it is really easy to create new characters, as they just have to be instantiated in the constructor method of the game class.

The Walk-Through

There are many ways of completing the game. One way is by entering the following command in order.

1. Go north
2. Go east
3. Grab ammo
4. back
5. Go north
6. Explore
7. Grab gun
8. Go up
9. Grab key
10. Back
11. Go south
12. Go down
13. Open west
14. Go west
15. Kill kryzko or Give kryzko poison

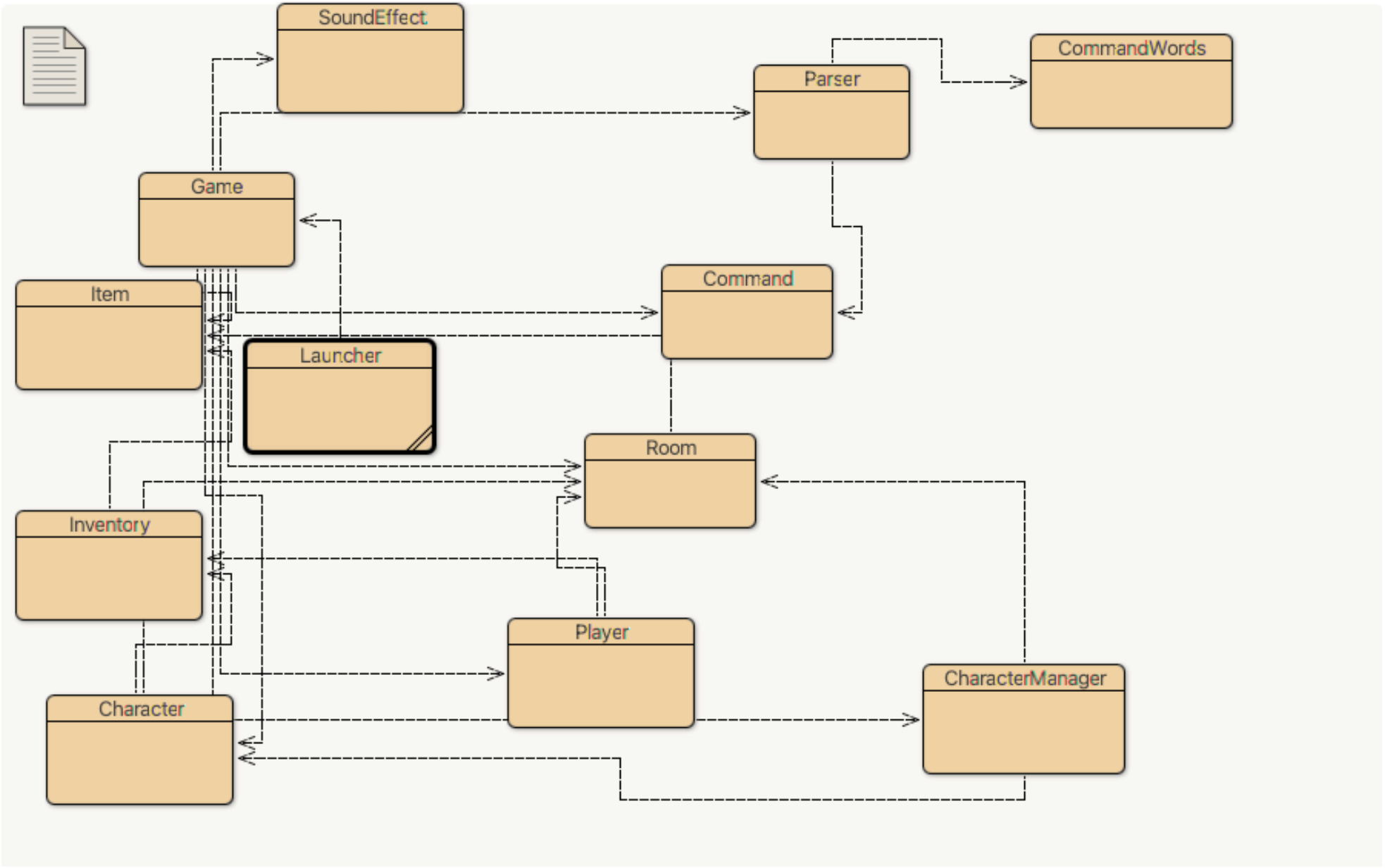
If the guard is encountered at any time during the game:

16. Kill guard or give guard poison

Known Bugs

One known bug is that when the back command is executed after using the transporter room. The player is transported to the attic not to the transporter room and teleported back again randomly.

There are no other known bugs in the game.


SoundEffect
CommandWords
Parser
Game
Command
Item
Launcher
Room
Inventory
Player
CharacterManager
Character

```
import java.io.*;
import java.net.URL;
import javax.sound.sampled.*;
/**
 * Class Inventory - the inventory of a player.
 *
 * This class is part of the James Bond application.
 *
 * Each player has an inventory. The inventory consists of an player
 * object (like a bag) that stores the items that the player grabs.
 *
 * The inventory can hold items up until a certain amount of weight.
 *
 * @author Unkown
 * @version 2021.11.28
 */
    public class SoundEffect {

        // Nested class for specifying volume
        public enum Volume {MUTE, MEDIUM}

        public Volume volume = Volume.MEDIUM;

        // Each sound effect has its own clip, loaded with its own sound file.
        private Clip clip;

        // Constructor to construct each element of the enum with its own sound
file.
        public SoundEffect(String soundFileName, double decibelsToReduce) {
           try {
              // Use URL (instead of File) to read from disk and JAR.
              URL url = this.getClass().getResource(soundFileName);
              // Set up an audio input stream piped from the sound file.
               assert url != null;
               AudioInputStream audioInputStream =
AudioSystem.getAudioInputStream(url);
              // Get a clip resource.
              clip = AudioSystem.getClip();
              // Open audio clip and load samples from the audio input stream.
              clip.open(audioInputStream);
              if (!soundFileName.equals("Kryzko.wav")) {
                  FloatControl gainControl = (FloatControl)
clip.getControl(FloatControl.Type.MASTER_GAIN);
                  gainControl.setValue(Float.parseFloat(-decibelsToReduce + "f"));
              }
           } catch (UnsupportedAudioFileException | IOException |
LineUnavailableException e) {
              e.printStackTrace();
           }
        }

        // Play or Re-play the sound effect from the beginning, by rewinding.
        public void play(Boolean loop) {
           if (volume != Volume.MUTE) {
              if (clip.isRunning())
```

```
                clip.stop();   // Stop the player if it is still running
              clip.setFramePosition(0); // rewind to the beginning
              clip.start();     // Start playing
              if(loop)//Loop if loop parameter is true
                   clip.loop(Clip.LOOP_CONTINUOUSLY);
           }

       }

       public void stop() //stop playing and rewind to be played again from the
beginning
       {
           clip.stop();
           clip.setFramePosition(0);
       }

    }
```

```java
/**
 * Class Player
 *
 * This class is part of the James Bond application.
 *
 * The player is the main character of the game.
 * He has several fields, his current room, previous room and his inventory.
 *
 * He also can be killed, if so he loses the game.
 *
 * @author Aymen Berbache
 * @version 2021.11.28
 */

public class Player
{
    private Room currentRoom;
    private Room previousRoom;
    private final Inventory inventory;
    private boolean isKilled;

    /**
     * Creates the player object. At first, the player has no previous room and an
empty inventory.
     */
    public  Player()
    {
        inventory = new Inventory();
        previousRoom = null;
        isKilled = false;
    }

    /**
     * @return The inventory of the player.
     */
    public Inventory getInventory()
    {
        return inventory;
    }

    /**
     * @return The current room of the player.
     */
    public Room getCurrentRoom()
    {
        return currentRoom;
    }

    /**
     * @return The previous room of the player.
     */
    public Room getPreviousRoom()
    {
        return previousRoom;
```

```
    }

    /**
     * Change the player current room.
     * @param destination The next room.
     */
    public void setCurrentRoom(Room destination)
    {
        currentRoom = destination;
    }

    /**
     *  Change the previous room of the player.
     * @param destination The next room.
     */
    public void setPreviousRoom(Room destination)
    {
        previousRoom = destination;
    }

    /**
     * @return true if the player has been killed, false if not.
     */
    public boolean isKilled(){return isKilled;}

    /**
     * Kill the player.
     */
    public void kill(){
        if(this.getInventory().getItem("gun") == null &&
this.getInventory().getItem("knife") == null){
            System.out.println("\nYou had no weapons on you!");
        }
        else {
            System.out.println("\nYou had no ammunition on you!");
        }
        System.out.println("You have thus been killed. Mission failed. RIP agent
007.");
        isKilled = true;
    }
}
```

```java
/**
 * Class Character - a character in an adventure game.
 *
 * This class is part of the James Bond application.
 *
 * A "Character" represents persons or animals. Characters are in rooms.
 * Characters can move around by themselves.
 *
 * Each character has a name, a description, its own inventory and a current room
location.
 *
 * @author Aymen Berbache
 * @version 2021.11.28
 */

public class Character
{
    private final String name;
    private final String description;
    private final Inventory inventory;
    private boolean isKilled;
    private Room currentRoom;

    /**
     * Contstructor that creates a character
     * @param name The name of the character.
     * @param description The description of the character.
     */
    public Character(String name, String description)
    {
        this.name = name;
        this.description = description;
        isKilled = false;
        inventory = new Inventory();
    }

    /**
     * Accessor method that returns the current room of the character.
     * @return The character current location.
     */
    public Room getCurrentRoom()
    {
        return currentRoom;
    }

    /**
     * Accessor method that returns the name of a character.
     * @return The name of the character
     */
    public String getName()
    {
        return name;
    }
```

```
    /**
     * Accessor method that returns the description of the character.
     * @return Something like "The guard of kryzko".
     */
    public String getDescription()
    {
        return description;
    }

    /**
     * Mutator method that set the initial room of the character when assigned a
room.
     * @param destination The room where the character is at the beginning of the
game.
     */
    public void setCurrentRoom(Room destination)
    {
        currentRoom = destination;
    }


    /**
     * Kill a character.
     */
    public void kill()
    {
        isKilled = true;
    }

    /**
     * Accessor method that show wether or not a character has been killed.
     * @return true if the character is dead, false if not.
     */
    public boolean isKilled(){return isKilled;}

    /**
     * @return The inventory of the character.
     */
    public Inventory getInventory()
    {
        return inventory;
    }

}
```

```java
import java.util.HashMap;
/**
 * Class Character Manager.
 *
 * This class is part of the James Bond application.
 *
 * This class manages the characters. That is to say, set their position in the
map
 * and allow them to move aroud the map.
 *
 * Each character has a name, a description, its own inventory and a current room
location.
 *
 * @author Aymen Berbache
 * @version 2021.11.28
 */
public class CharacterManager
{

    private final HashMap<Room,Character> charactersInRoom;
    private final HashMap<String,Character> charactersMap;

    public CharacterManager(){
        charactersInRoom = new HashMap<>();
        charactersMap = new HashMap<>();
    }

    /**
     * Set the character room.
     * @param room  Room containing the character.
     * @param character Character concerned.
     */
    public void setCharacterInRoom(Room room, Character character)
    {
        charactersMap.put(character.getName(),character);
        charactersInRoom.put(room,character);
        character.setCurrentRoom(room);

    }

    /**
     * @param characterName The nam of the character we want.
     * @return The character object.
     */
    public Character getCharacter(String characterName)
    {
        return charactersMap.get(characterName);
    }

    /**
     * Remove character from room.
     * @param room
     * @param character
     */
    public void removeCharacterFromRoom(Room room, Character character)
```

```
    {
        charactersInRoom.remove(room, character);
    }

    /**
     * Command that make the character moves randomly exit from exit.
     * @param characterToMove
     */
    public void moveRandomly(Character characterToMove)
    {
        if (!characterToMove.isKilled()) {
            // Create local variable that holds the random exit generated
            Room randomRoom =
characterToMove.getCurrentRoom().generateRandomExit();

removeCharacterFromRoom(characterToMove.getCurrentRoom(),characterToMove);
            setCharacterInRoom(randomRoom,characterToMove);
            System.out.println(characterToMove.getDescription()+" is " +
characterToMove.getCurrentRoom().getShortDescription() + ".");
        }
    }
}
```

```
import java.util.HashMap;
/**
 * This class is part of the James Bond 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  Aymen Berbache, Michael Kölling and David J. Barnes
 * @version 2021.11.28
 */

public class CommandWords {

    // a constant Map that holds all valid command words and their description
    private static final HashMap<String,String> commandWords = new HashMap<>();

    static {
        commandWords.put("go","Allows you to navigate through the rooms.");
        commandWords.put("quit","Allows you to leave the game.");
        commandWords.put("help","Displays some help statements.");
        commandWords.put("explore","Displays all items available in the room you
want to explore.");
        commandWords.put("grab","Allows you to grab an item and put it in your
inventory.");
        commandWords.put("throw","Allows you to get rid of an item in your
inventory.");
        commandWords.put("show","Displays the content of your inventory.");
        commandWords.put("back","Takes you in the room you were in previously.");
        commandWords.put("weight","Displays the weight of an item.");
        commandWords.put("open","Try to open a locked door.");
        commandWords.put("kill","Kills the character you are interacting with.");
        commandWords.put("give","Give item to a character. (give 'character'
'item')");
    }

    /**
     * Constructor - initialise the command words.
     */
    public CommandWords() {
        //Nothing...
    }


    /**
     * 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 < commandWords.size(); i++) {
            if (commandWords.keySet().toArray()[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() {
        System.out.printf("%-20s%s","Command","Description");
        System.out.println();
        for (String command : commandWords.keySet()) {
            System.out.printf("%-20s%s\n",command, commandWords.get(command));
        }
        System.out.println();
    }
}
```

```java
import java.util.Scanner;

/**
 * This class is part of the James Bond 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 two or three word command. It returns the
command
 * as an object of class Command.
 *
 * The parser has a set of known command words. It checks user input against
 * the known commands, and if the input is not one of the known commands, it
 * returns a command object that is marked as an unknown command.
 *
 * @author  Aymen Berbache, Michael Kölling and David J. Barnes
 * @version 2021.11.30
 */
public class Parser
{
    private final CommandWords commands;  // holds all valid command words
    private final 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().trim().toLowerCase();

        // Find up to two words on the line.
        Scanner tokenizer = new Scanner(inputLine);
        if(tokenizer.hasNext()) {
            word1 = tokenizer.next();      // get first word
            if(tokenizer.hasNext()) {
                word2 = tokenizer.next();      // get second word
                if(tokenizer.hasNext()) {
                    word3 = tokenizer.next();      // get second word
                // note: we just ignore the rest of the input line
```

```
                }
            }
        }

        // Now check whether this word is known. If so, create a command
        // with it. If not, create a "null" command (for unknown command).
        if(commands.isCommand(word1)) {
            return new Command(word1, word2, word3);
        }
        else {
            return new Command(null, word2, word3);
        }
    }

    /**
     * Print out a list of valid command words.
     */
    public void showCommands()
    {
        commands.showAll();
    }
}
```

```java
/**
 * Class Item
 *
 * This class is part of the James Bond application.
 *
 * An item is any tangible object that can be found in the rooms of the game.
 * Each item has a name, a weight and a description.
 *
 * An item can be picked up, thrown away and carried around the rooms by the
player.
 *
 * @author Aymen Berbache
 * @version 2021.11.28
 */

public class Item {

    private final String itemName;
    private final String description;
    private final int weight;
    private final boolean canBeCarried;

    /**
     * Creates the item.
     * @param itemName The name of the item.
     * @param description The description of the item.
     * @param weight The weight of the item.
     */
    public Item(String itemName, String description, int weight, boolean
canBeCarried)
    {
        this.itemName = itemName;
        this.description = description;
        this.weight = weight;
        this.canBeCarried = canBeCarried;
    }

    /**
     * @return The description of the item. Something like that :"butcher knife".
     */
    public String getItemDescription()
    {
        return description;
    }

    /**
     * @return The name of the item.
     */
    public String getItemName()
    {
        return itemName;
    }

    /**
     * @return The weight of the item
```