1. Start with the bad **Zuul** game and refactor it as discussed in the lectures. If you are doing mole burrows instead of rooms, you can change the variable names as needed. The bored can prepare a multi-lingual version and use enums.

The first thing we changed was the way we define exits and locations. Therefore, we needed to change the location class. We added a HashMap which will save the custom name of the exit with the location that can be accessed in this direction.

```
public class Location
{
    private HashMap<String, Location> exits;
    private String description;
```

```
public void addExit(String name, Location exit)
{
    exits.put(name, exit);
}
```

```
private void createLocations()
{
    Location trackN, trackW, trackS, trackE, soccer, longJump, shed;

    // create the Locations
    trackN = new Location("on the northern part of the tartan track");
    trackW = new Location("on the western part of the tartan track");
    trackS = new Location("on the southern part of the tartan track");
    trackE = new Location("on the eastern part of the tartan track");
    soccer = new Location("on the soccerfield");
    longJump = new Location("in the long jump area with a sandbox:)");
    shed = new Location("at the shed");

    // initialise Location exits
    trackN.addExit("west", trackW);
    trackN.addExit("east", trackE);
    trackN.addExit("south", soccer);

    trackW.addExit("north", trackN);
    trackW.addExit("east", soccer);
    trackW.addExit("south", trackS);

    trackE.addExit("north", trackN);
    trackE.addExit("west", soccer);
    trackE.addExit("south", trackS);

    trackS.addExit("north", soccer);
    trackS.addExit("east", trackE);
    trackS.addExit("west", trackW);
    trackS.addExit("south", longJump);

    soccer.addExit("north", trackN);
    soccer.addExit("east", trackE);
    soccer.addExit("west", trackW);
    soccer.addExit("south", trackS);

    longJump.addExit("north", trackS);
    longJump.addExit("east", shed);

    shed.addExit("west", longJump);

    currentLocation = soccer;  // start game outside
}
```

Next, we wanted to implement “enum”. Therefore, we created a new enum class in which we defined all valid commands.

```
public enum CommandWords
{
    // a constant array that holds all valid command words
    GO, HELP, QUIT, LOOK, OPEN, SLEEP, UNKNOWN;

}
```

Next, we had to adapt a few methods in the other classes to make the game use these commands instead of given Strings. We added a new HashMap inside the CommandWord class. This map saves a String (used to control the game) and the right command for each of them.

```
public class CommandWord
{
    HashMap<String, CommandWords> commands = new HashMap<>();

    /**
     * Constructor - initialise the command words.
     */
    CommandWord()
    {
        commands.put("go", CommandWords.GO);
        commands.put("help", CommandWords.HELP);
        commands.put("quit", CommandWords.QUIT);
        commands.put("look", CommandWords.LOOK);
        commands.put("open", CommandWords.OPEN);
        commands.put("sleep", CommandWords.SLEEP);
    }
```

We added a method which takes a String as its parameter (user’s input) and returns the enum command if it is in the hashmap. If its not in there the method returns ‘UNKNOWN’. We need that in case the user uses a wrong command.

```
public CommandWords getEnum(String word){
    if(commands.containsKey(word)) return commands.get(word);
    else return CommandWords.UNKNOWN;
}
```

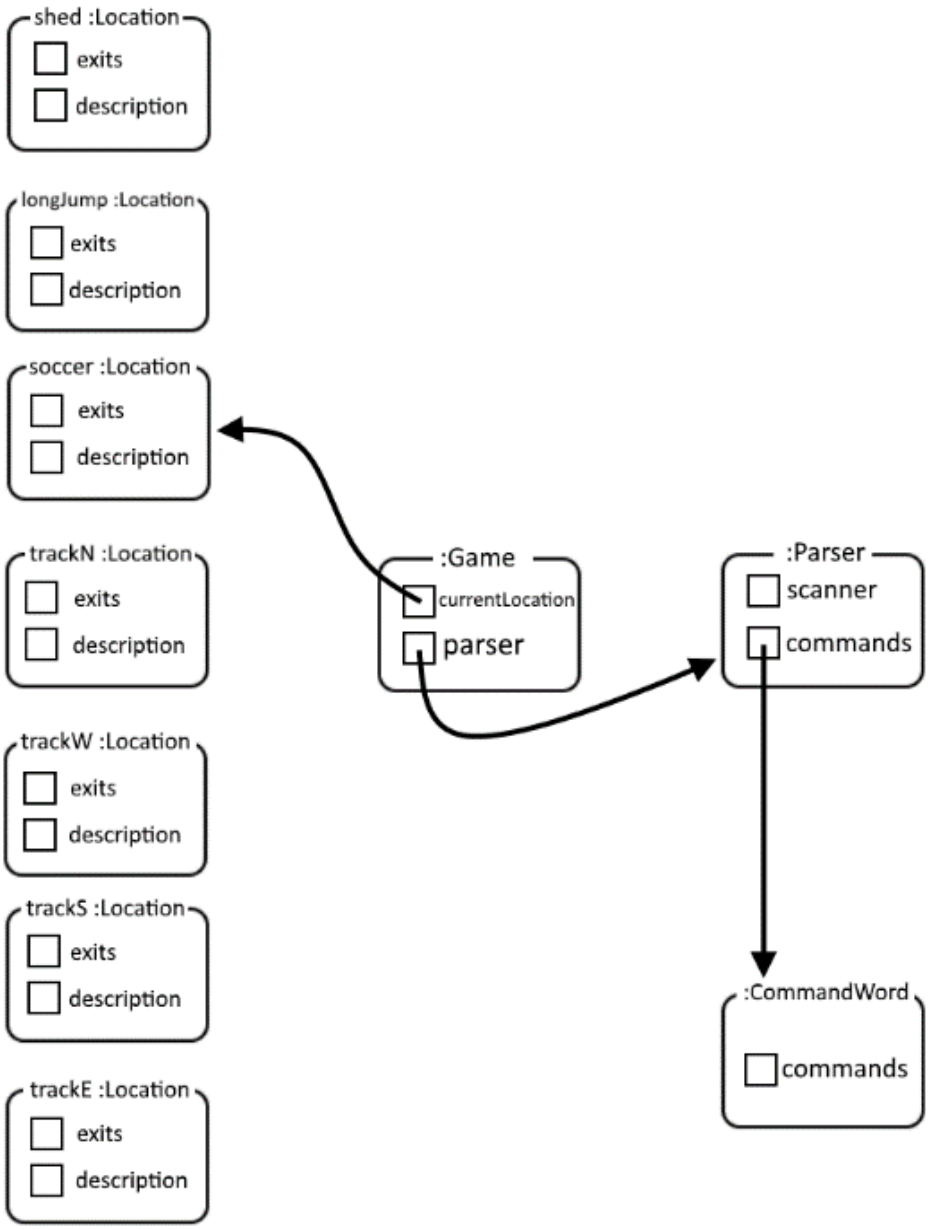
To enable the enum commands and make them work we needed to change the parser-class a bit as well. The first word (“word1”) will always be an enum command (GO, LOOK, HELP, ...) and second one is always a String (GO “east”). That’s why we changed the type of the first input to CommandWords and used our new created method “getEnum()” to return the fitting enum command and save it in “word1”.

```
public Command getCommand()
{
    String inputLine;   // will hold the full input line
    CommandWords word1 = null;
    String word2 = null;

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

    inputLine = reader.nextLine();

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

```
private boolean processCommand(Command command)
{
    boolean wantToQuit = false;

    CommandWords commandWord = command.getCommandWord();
    switch (commandWord){
        case HELP:
            printHelp();
            break;

        case GO:
            goLocation(command);
            break;

        case QUIT:
            wantToQuit = quit(command);
            break;

        case LOOK:
            look();
            break;

        case OPEN:
            open(command);
            break;

        case SLEEP:
            sleep();
            break;

        case UNKNOWN:
            System.out.println("I don't know what you mean...");
            return false;
    }
```

The last thing we had to do was to change the processCommand() method in the game class. This method chooses the response for each command. We changed the “if else(s)” to a switch-case to check the command using cases instead of an else-if statement for every single command.

2. Draw an object diagram showing the state of your system just after it has been started. Does it change if you issue a "go" command?



3. Add a "look" command to your game.

Add an additional command (such as "eat", which for now just prints out "You have eaten now and are not hungry any more". In the next exercise, when we have added items, you can make it so that you can only eat if you have found food.

For this exercise, we first added the “look” command to the CommandWords enum and added a case to the processCommand method which calls the method look. Then we added a new method called look that prints out the long description of the current location.

```
public enum CommandWords
{
    // a constant array that holds all valid command words
    GO, HELP, QUIT, LOOK, OPEN, SLEEP, UNKNOWN;

}
```

```
commands.put("look", CommandWords.LOOK);
commands.put("open", CommandWords.OPEN);
```

For the next exercise (which is included here), we added the “open” command to the enum and added another case that calls the open method and passes the command to it. Then we added the new method open to the Game class, which checks if the command entered has a second keyword. If it does, the method does nothing right now, but if it doesn’t, the method prints out “What do you want to open?”.

```
case LOOK:
    look();
    break;

case OPEN:
    open(command);
    break;
```

```
/**
 * Look around. Prints out the long description of the current location.
 */
private void look(){
    System.out.println(currentLocation.getLongDescription());
}
```

```
/**
 * Try to open something.
 * @param command The command that triggered this method call.
 */
private void open(Command command) {
    if (!command.hasSecondWord()) {
        System.out.println("What do you want to open?");
    }
}
```

4. Implement an improved version of printing out the command words.

Since we added a HashMap to keep track of the commands in exercise 1, it was very easy to add a refined getCommands method. We rewrote the method to loop through the HashMap, adding each command to a String separated by a comma. After the for-each loop, the last comma in the String is replaced by an empty String. The resulting String is then returned.

```
public String getCommands(){
    String cmd ="";
    for(String command : commands.keySet()){
        cmd += command + ", ";
    }
    cmd = cmd.replaceAll(", $", "");
    return cmd;
}
```

5. Add another command - did you have to change the Game class? Why or why not?

We added another command called sleep to the CommandWords enum and added a reference to it to the commands HashMap in the CommandWord class. Then we had to change the Game class to add a new case to the processCommand method which calls the sleep method. We also had to add the sleep method to the Game class to provide code to be executed when the sleep command is entered.

Evaluation

David

I'm really excited to start programming on our own game! We worked pretty fast together and finished the tasks fast. Using regular expressions, we learned about in CS was fun. I'm excited for the next lab:)

Luis

Programming our own game is a very exciting task, though it wasn't easy to remember everything we had changed in the lecture before holidays. I'm really looking forward to extending the game's scale by adding items or enemies.

```
1
2
3  /**
4   *  This class is the main class of the "World of Zuul" application.
5   *  "World of Zuul" is a very simple, text based adventure game.  Users
6   *  can walk around some scenery. That's all. It should really be extended
7   *  to make it more interesting!
8   *
9   *  To play this game, create an instance of this class and call the "play"
10  *  method.
11  *
12  *  This main class creates and initialises all the others: it creates all
13  *  Locations, creates the parser and starts the game.  It also evaluates and
14  *  executes the commands that the parser returns.
15  *
16  * @author  David Panagiotopulos and Luis Hankel
17  * @version 2018.01.09
18  *
19  */
20
21 public class Game
22 {
23     private Parser parser;
24     private Location currentLocation;
25
26     /**
27      * Create the game and initialise its internal map.
28      */
29     public Game()
30     {
31         createLocations();
32         parser = new Parser();
33     }
34
35     /**
36      * Create all the Locations and link their exits together.
37      */
38     private void createLocations()
39     {
40         Location trackN, trackW, trackS, trackE, soccer, longJump, shed;
41
42         // create the Locations
43         trackN = new Location("on the northern part of the tartan track");
44         trackW = new Location("on the western part of the tartan track");
45         trackS = new Location("on the southern part of the tartan track");
46         trackE = new Location("on the eastern part of the tartan track");
47         soccer = new Location("on the soccerfield");
48         longJump = new Location("in the long jump area with a sandbox:)");
49         shed = new Location("at the shed");
50
```

```
51          // initialise Location exits
52          trackN.addExit("west", trackW);
53          trackN.addExit("east", trackE);
54          trackN.addExit("south", soccer);
55
56          trackW.addExit("north", trackN);
57          trackW.addExit("east", soccer);
58          trackW.addExit("south", trackS);
59
60          trackE.addExit("north", trackN);
61          trackE.addExit("west", soccer);
62          trackE.addExit("south", trackS);
63
64          trackS.addExit("north", soccer);
65          trackS.addExit("east", trackE);
66          trackS.addExit("west", trackW);
67          trackS.addExit("south", longJump);
68
69          soccer.addExit("north", trackN);
70          soccer.addExit("east", trackE);
71          soccer.addExit("west", trackW);
72          soccer.addExit("south", trackS);
73
74          longJump.addExit("north", trackS);
75          longJump.addExit("east", shed);
76
77          shed.addExit("west", longJump);
78
79          currentLocation = soccer;  // start game outside
80      }
81
82      /**
83       *  Main play routine.  Loops until end of play.
84       */
85      public void play()
86      {
87          printWelcome();
88
89          // Enter the main command loop.  Here we repeatedly read commands and
90          // execute them until the game is over.
91
92          boolean finished = false;
93          while (! finished) {
94              Command command = parser.getCommand();
95              finished = processCommand(command);
96          }
97          System.out.println("Thank you for playing.  Good bye.");
98      }
99
100     /**
```

```
101     * Given a command, process (that is: execute) the command.
102     * @param command The command to be processed.
103     * @return true If the command ends the game, false otherwise.
104     */
105    private boolean processCommand(Command command)
106    {
107        boolean wantToQuit = false;
108
109        CommandWords commandWord = command.getCommandWord();
110        switch (commandWord){
111            case HELP:
112                printHelp();
113                break;
114
115            case GO:
116                goLocation(command);
117                break;
118
119            case QUIT:
120                wantToQuit = quit(command);
121                break;
122
123            case LOOK:
124                look();
125                break;
126
127            case OPEN:
128                open(command);
129                break;
130
131            case SLEEP:
132                sleep();
133                break;
134
135            case UNKNOWN:
136                System.out.println("I don't know what you mean...");
137                return false;
138        }
139
140        // if (commandWord.equals("help"))
141            // printHelp();
142        // else if (commandWord.equals("go"))
143            // goLocation(command);
144        // else if (commandWord.equals("quit"))
145            // wantToQuit = quit(command);
146        // else if (commandWord.equals("look"))
147            // look();
148        // else if (commandWord.equals("open"))
149            // open(command);
150        // else if (commandWord.equals("sleep"))
```

```
151             // sleep();
152
153
154         return wantToQuit;
155     }
156
157     /**
158      * Print out the opening message for the player.
159      */
160     private void printWelcome()
161     {
162         System.out.println();
163         System.out.println("Welcome to SportyZombies!");
164         System.out.println("SportyZombies is a new, incredible horror adventure
    game.");
165         System.out.println("Type 'help' if you need help.");
166         System.out.println();
167         System.out.println(currentLocation.getLongDescription());
168     }
169
170     // implementations of user commands:
171
172     /**
173      * Print out some help information.
174      * Here we print some stupid, cryptic message and a list of the
175      * command words.
176      */
177     private void printHelp()
178     {
179         System.out.println("You are lost. But you are alone. A horde of zombies");
180         System.out.println("is running towards you.");
181         System.out.println();
182         System.out.println("Your command words are:");
183         System.out.println(parser.getCommands());
184     }
185
186     /**
187      * Try to go to one direction. If there is an exit, enter
188      * the new Location, otherwise print an error message.
189      */
190     private void goLocation(Command command)
191     {
192         if(!command.hasSecondWord()) {
193             // if there is no second word, we don't know where to go...
194             System.out.println("Go where?");
195             return;
196         }
197
198         String direction = command.getSecondWord();
199
```

```
200         // Try to leave current Location.
201         Location nextLocation = null;
202
203         nextLocation = currentLocation.getExit(direction);
204
205         if (nextLocation == null) {
206             System.out.println("There is no door!");
207         }
208         else {
209             currentLocation = nextLocation;
210             System.out.println(currentLocation.getLongDescription());
211         }
212     }
213
214     /**
215      * "Quit" was entered. Check the rest of the command to see
216      * whether we really quit the game.
217      * @return true, if this command quits the game, false otherwise.
218      */
219     private boolean quit(Command command)
220     {
221         if(command.hasSecondWord()) {
222             System.out.println("Quit what?");
223             return false;
224         }
225         else {
226             return true;  // signal that we want to quit
227         }
228     }
229
230     /**
231      * Look around. Prints out the long description of the current location.
232      */
233     private void look(){
234         System.out.println(currentLocation.getLongDescription());
235     }
236
237     /**
238      * Try to open something.
239      * @param command The command that triggered this method call.
240      */
241     private void open(Command command) {
242         if (!command.hasSecondWord()) {
243             System.out.println("What do you want to open?");
244         }
245     }
246     private void sleep(){
247         System.out.println("You've slept"+"\n"+"You feel soooo good now!");
248     }
249 }
```

```java
import java.util.HashMap;
/**
 * Class Location - a Location in an adventure game.
 *
 * This class is part of the "World of Zuul" application.
 * "World of Zuul" is a very simple, text based adventure game.
 *
 * A "Location" represents one location in the scenery of the game.  It is
 * connected to other Locations via exits.  The exits are labelled north,
 * east, south, west.  For each direction, the Location stores a reference
 * to the neighboring Location, or null if there is no exit in that direction.
 *
 * @author  David Panagiotopulos and Luis Hankel
 * @version 2018.01.09
 */
public class Location
{
    private HashMap<String, Location> exits;
    private String description;

    /**
     * Create a Location described "description". Initially, it has
     * no exits. "description" is something like "a kitchen" or
     * "an open court yard".
     * @param description The Location's description.
     */
    public Location(String description)
    {
        this.description = description;
        exits = new HashMap<>();
    }

    /**
     * Define the exits of this Location.  Every direction either leads
     */
    public void addExit(String name, Location exit)
    {
        exits.put(name, exit);
    }

    /**
     * @return The description of the Location.
     */
    public String getDescription()
    {
        return description;
    }

    /**
     * @return The long description of the Location.
```

```
51      */
52      public String getLongDescription()
53      {
54          return "You are " + getDescription() + "\n" + getExits();
55      }
56
57      public String getExits(){
58          String allExits = "Exits: ";
59          for(String name : exits.keySet()){
60              allExits += name + ", ";
61          }
62          allExits = allExits.replaceAll(", $","");
63
64          return allExits;
65      }
66
67      public Location getExit(String exit){
68          return exits.get(exit);
69      }
70  }
71
```

```java
/**
 * This class is part of the "World of Zuul" application.
 * "World of Zuul" 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 two strings: a command word and a second
 * word (for example, if the command was "take map", then the two strings
 * obviously are "take" and "map").
 *
 * The way this is used is: Commands are already checked for being valid
 * command words. If the user entered an invalid command (a word that is not
 * known) then the command word is <null>.
 *
 * If the command had only one word, then the second word is <null>.
 *
 * @author  David Panagiotopulos and Luis Hankel
 * @version 2018.01.09
 */

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

    /**
     * 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(CommandWords firstWord, String secondWord)
    {
        commandWord = firstWord;
        this.secondWord = secondWord;
    }

    /**
     * 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 CommandWords getCommandWord()
    {
        return commandWord;
    }

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

```
51          */
52      public String getSecondWord()
53      {
54          return secondWord;
55      }
56
57      /**
58       * @return true if this command was not understood.
59       */
60      public boolean isUnknown()
61      {
62          return (commandWord == null);
63      }
64
65      /**
66       * @return true if the command has a second word.
67       */
68      public boolean hasSecondWord()
69      {
70          return (secondWord != null);
71      }
72  }
73
74
```

```
1  import java.util.Scanner;
2  import java.util.StringTokenizer;
3
4  /**
5   * This class is part of the "World of Zuul" application.
6   * "World of Zuul" is a very simple, text based adventure game.
7   *
8   * This parser reads user input and tries to interpret it as an "Adventure"
9   * command. Every time it is called it reads a line from the terminal and
10  * tries to interpret the line as a two word command. It returns the command
11  * as an object of class Command.
12  *
13  * The parser has a set of known command words. It checks user input against
14  * the known commands, and if the input is not one of the known commands, it
15  * returns a command object that is marked as an unknown command.
16  *
17  * @author  David Panagiotopulos and Luis Hankel
18  * @version 2018.01.09
19  */
20 public class Parser
21 {
22     private CommandWord commands;  // holds all valid command words
23     private Scanner reader;         // source of command input
24
25     /**
26      * Create a parser to read from the terminal window.
27      */
28     public Parser()
29     {
30         commands = new CommandWord();
31         reader = new Scanner(System.in);
32     }
33
34     /**
35      * @return The next command from the user.
36      */
37     public Command getCommand()
38     {
39         String inputLine;   // will hold the full input line
40         CommandWords word1 = null;
41         String word2 = null;
42
43         System.out.print("> ");     // print prompt
44
45         inputLine = reader.nextLine();
46
47         // Find up to two words on the line.
48         Scanner tokenizer = new Scanner(inputLine);
49         if(tokenizer.hasNext()) {
50             word1 = commands.getEnum(tokenizer.next());      // get first word
```

```
51              if(tokenizer.hasNext()) {
52                  word2 = tokenizer.next();      // get second word
53                  // note: we just ignore the rest of the input line.
54              }
55          }
56          return new Command(word1, word2);
57      }
58
59      public String getCommands(){
60          return commands.getCommands();
61      }
62  }
63
```

```
/**
 * This class is part of the "World of Zuul" application.
 * "World of Zuul" 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  David Panagiotopulos and Luis Hankel
 * @version 2018.01.09
 */

public enum CommandWords
{
    // a constant array that holds all valid command words
    GO, HELP, QUIT, LOOK, OPEN, SLEEP, UNKNOWN;

}

```

```java
import java.util.HashMap;
/**
 * Write a description of class CommandWord here.
 *
 * @author  David Panagiotopulos and Luis Hankel
 * @version 2018.01.09
 */
public class CommandWord
{
    HashMap<String, CommandWords> commands = new HashMap<>();

    /**
     * Constructor - initialise the command words.
     */
    CommandWord()
    {
        commands.put("go", CommandWords.GO);
        commands.put("help", CommandWords.HELP);
        commands.put("quit", CommandWords.QUIT);
        commands.put("look", CommandWords.LOOK);
        commands.put("open", CommandWords.OPEN);
        commands.put("sleep", CommandWords.SLEEP);
    }

    /**
     * Check whether a given String is a valid command word.
     * @return true if a given string is a valid command,
     * false if it isn't.
     */
    public boolean isCommand(String cmd)
    {
        return commands.containsKey(cmd);
    }

    public String getCommands(){
        String cmd ="";
        for(String command : commands.keySet()){
            cmd += command + ", ";
        }
        cmd = cmd.replaceAll(", $", "");
        return cmd;
    }

    public CommandWords getEnum(String word){
        if(commands.containsKey(word)) return commands.get(word);
        else return CommandWords.UNKNOWN;
    }
}
```