Exercise 'text based game'

```
1  /**
2   * This class is part of the "World of Zuul" application.
3   * "World of Zuul" is a very simple, text based adventure game.
4   *
5   * This class holds information about a command that was issued by the
        user.
6   * A command currently consists of two parts: a CommandWord and a string
7   * (for example, if the command was "take map", then the two parts
8   * are TAKE and "map").
9   *
10  * The way this is used is: Commands are already checked for being valid
11  * command words. If the user entered an invalid command (a word that is
        not
12  * known) then the CommandWord is UNKNOWN.
13  *
14  * If the command had only one word, then the second word is <null>.
15  *
16  * @author  Michael Ã¶Kling and David J. Barnes
17  * @version 2011.08.10
18  */
19
20 public class Command
21 {
22     private CommandWord commandWord;
23     private String secondWord;
24
25     /**
26      * Create a command object. First and second words must be supplied,
            but
27      * the second may be null.
28      * @param commandWord The CommandWord. UNKNOWN if the command word
29      *                    was not recognised.
30      * @param secondWord The second word of the command. May be null.
31      */
32     public Command(CommandWord commandWord, String secondWord)
33     {
34         this.commandWord = commandWord;
35         this.secondWord = secondWord;
36     }
37
38     /**
39      * Return the command word (the first word) of this command.
40      * @return The command word.
41      */
42     public CommandWord getCommandWord()
43     {
44         return commandWord;
45     }
46
47     /**
48      * @return The second word of this command. Returns null if there was
            no
49      * second word.
```

```
50          */
51      public String getSecondWord()
52      {
53          return secondWord;
54      }
55
56      /**
57       * @return true if this command was not understood.
58       */
59      public boolean isUnknown()
60      {
61          return (commandWord == CommandWord.UNKNOWN);
62      }
63
64      /**
65       * @return true if the command has a second word.
66       */
67      public boolean hasSecondWord()
68      {
69          return (secondWord != "");
70      }
71  }
```

```
1   /**
2    * Representations for all the valid command words for the game
3    * along with a string in a particular language.
4    *
5    * @author  Michael Ã¶Klling and David J. Barnes
6    * @version 2011.08.10
7    */
8   public enum CommandWord
9   {
10      // A value for each command word along with its
11      // corresponding user interface string.
12      GO("go"), QUIT("quit"), HELP("help"), UNKNOWN("?"), UNLOCK("unlock"),
            ASK("ask"), TWEET("tweet");
13
14      // The command string.
15      private String commandString;
16
17      /**
18       * Initialise with the corresponding command string.
19       * @param commandString The command string.
20       */
21      CommandWord(String commandString)
22      {
23          this.commandString = commandString;
24      }
25
26
27
28
29      /**
30       * @return The command word as a string.
31       */
32      public String toString()
```

```
33      {
34          return commandString;
35      }
36  }
```

```
1   import java.util.HashMap;
2
3   /**
4    * This class is part of the "World of Zuul" application.
5    * "World of Zuul" is a very simple, text based adventure game.
6    *
7    * This class holds an enumeration of all command words known to the game.
8    * It is used to recognise commands as they are typed in.
9    *
10   * @author  Michael Ã¶Klling and David J. Barnes
11   * @version 2011.08.10
12   */
13
14  public class CommandWords
15  {
16      // A mapping between a command word and the CommandWord
17      // associated with it.
18      private HashMap<String, CommandWord> validCommands;
19
20      /**
21       * Constructor - initialise the command words.
22       */
23      public CommandWords()
24      {
25          validCommands = new HashMap<String, CommandWord>();
26          for(CommandWord command : CommandWord.values()) {
27              if(command != CommandWord.UNKNOWN) {
28                  validCommands.put(command.toString(), command);
29              }
30          }
31      }
32
33      /**
34       * Find the CommandWord associated with a command word.
35       * @param commandWord The word to look up.
36       * @return The CommandWord correspondng to commandWord, or UNKNOWN
37       *          if it is not a valid command word.
38       */
39      public CommandWord getCommandWord(String commandWord)
40      {
41          CommandWord command = validCommands.get(commandWord);
42          if(command != null) {
43              return command;
44          }
45          else {
46              return CommandWord.UNKNOWN;
47          }
48      }
49
50      /**
51       * Check whether a given String is a valid command word.
```

```
52          * @return true if it is, false if it isn't.
53          */
54         public boolean isCommand(String aString)
55         {
56             return validCommands.containsKey(aString);
57         }
58 
59         /**
60          * Print all valid commands to System.out.
61          */
62         public void showAll()
63         {
64             for(String command : validCommands.keySet()) {
65                 System.out.print(command + "  ");
66             }
67             System.out.println();
68         }
69 }
```

```
1 
2 public class Door {
3     private Room room;
4     private String password;
5 
6     public Door(Room room, String password)
7     {
8         this.room = room;
9         this.password = password;
10    }
11 
12    public Room getRoom()
13    {
14        return room;
15    }
16 
17    public boolean hasDoor()
18    {
19        if(room == null)
20        {
21            return false;
22        }
23        return true;
24    }
25 
26    /**
27     * Check if a room is password protected
28     * @return true if password was assigned, false otherwise.
29     */
30    public boolean hasPassword()
31    {
32        if(password != "")
33        {
34            return true;
35        }
36        else
37        {
```

```
38              return false;
39          }
40      }
41
42      /**
43       * Check whether the provided password equals to the password in the
             room.
44       * @param password
45       * @return
46       */
47      public boolean doUnlock(String password)
48      {
49          if(this.password.equals(password))
50          {
51              password ="";
52              return true;
53          }
54          else
55          {
56              return false;
57          }
58      }
59  }
```

```
1   import java.util.Random;
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   *  rooms, creates the parser and starts the game.  It also evaluates and
14   *  executes the commands that the parser returns.
15   *
16   * @author  Michael Ã¶Klling and David J. Barnes
17   * @version 2011.08.10
18   */
19
20  public class Game
21  {
22      private Parser parser;
23      private Room currentRoom;
24
25      /**
26       * Create the game and initialise its internal map.
27       */
28      public Game()
29      {
30          createRooms();
31          parser = new Parser();
32      }
```

```
33
34      /**
35       * Create all the rooms and link their exits together.
36       */
37
38      private Room outside, D, E, F, TekHogUnderground, D2, D3, D3_1,
            DCompLab;
39      private int tweetCount = 20;
40
41
42      private void createRooms()
43      {
44  //          Room outside, theater, pub, lab, office;
45
46
47
48          Person Jockum  = new Person("Jockum", x-> JockumResponse(x));
49
50          Person Armada = new Person("Armada", x-> "Please work for
                Armada!");
51
52          Person Jonas = new Person("Jonas", x-> JonasResponse(x));
53
54          outside = new Room("outside the main entrace of KTH");
55          D = new Room("in the D building");
56          D2 = new Room("on the second floor in D building.");
57          D3 = new Room("on the third florr in D building.");
58          D3_1 = new Room("in the lecture hall D1 where we've just started
                with single variable calculus. You are late.");
59
60          DCompLab = new Room("in the computer laboratory.");
61          E = new Room("in the E building");
62          F = new Room("in the F building");
63          TekHogUnderground = new Room("at Tekniska öHgskolan Underground
                station");
64
65          D3_1.setPerson(Jockum);
66          outside.setPerson(Armada);
67          E.setPerson(Jonas);
68          D.setPerson(Armada);
69
70          outside.setExit("down", TekHogUnderground);
71          outside.setExit("west", E, "komm14");
72          outside.setExit("east", D);
73          outside.setExit("north", F);
74
75          E.setExit("east", outside);
76          D.setExit("west", outside);
77          D.setExit("up", D2);
78
79          D2.setExit("up", D3);
80          D2.setExit("down", D);
81          D3.setExit("north", D3_1);
82          D3.setExit("south", DCompLab);
83          D3.setExit("down", D2);
84
```

```
85          D3_1.setExit("south", D3);
86          DCompLab.setExit("north", D3);
87
88
89
90          TekHogUnderground.setExit("up", outside);
91
92          currentRoom = TekHogUnderground;
93      }
94
95      /**
96       *  Main play routine.  Loops until end of play.
97       */
98      public void play()
99      {
100         printWelcome();
101
102         // Enter the main command loop.  Here we repeatedly read commands
                and
103         // execute them until the game is over.
104
105         boolean finished = false;
106         while (! finished) {
107             Command command = parser.getCommand();
108             finished = processCommand(command);
109         }
110         System.out.println("Thank you for playing.  Good bye.");
111     }
112
113     /**
114      * Print out the opening message for the player.
115      */
116     private void printWelcome()
117     {
118         System.out.println();
119         System.out.println("Welcome to Java Adventure at KTH");
120         System.out.println("Java Adventure at KTH is a new, incredibly
                interesting adventure game.");
121         System.out.println("Type '" + CommandWord.HELP + "' if you need
                help.");
122         System.out.println();
123         System.out.println(currentRoom.getLongDescription());
124     }
125
126     /**
127      * Given a command, process (that is: execute) the command.
128      * @param command The command to be processed.
129      * @return true If the command ends the game, false otherwise.
130      */
131     private boolean processCommand(Command command)
132     {
133         boolean wantToQuit = false;
134
135         CommandWord commandWord = command.getCommandWord();
136
137         switch (commandWord) {
```

```
138             case UNKNOWN:
139                 System.out.println("I don't know what you mean...");
140                 break;
141 
142             case HELP:
143                 printHelp();
144                 break;
145 
146             case GO:
147                 TransformUsers();
148                 goRoom(command);
149                 break;
150 
151             case QUIT:
152                 wantToQuit = quit(command);
153                 break;
154 
155             case ASK:
156                 doAsk(command);
157                 break;
158             case TWEET:
159                 tweetCount--;
160                 break;
161             case UNLOCK:
162                 System.out.println("What should we unlock?");
163                 break;
164         }
165         return wantToQuit;
166     }
167 
168     // implementations of user commands:
169 
170     /**
171      * Print out some help information.
172      * Here we print some stupid, cryptic message and a list of the
173      * command words.
174      */
175     private void printHelp()
176     {
177         System.out.println("You are lost. You are alone. You wander");
178         System.out.println("around at the university.");
179         System.out.println();
180         System.out.println("Your command words are:");
181         parser.showCommands();
182     }
183 
184 
185     private void doAsk(Command command)
186     {
187         if(!command.hasSecondWord()) {
188             // if there is no second word, we don't know where to go...
189             System.out.println("Ask what?");
190             return;
191         }
192 
193         if(currentRoom.getPerson() != null)
```

```
194         {
195             //System.out.println(command.getSecondWord());
196             System.out.println(currentRoom.getPerson().getResponse(command.getSecondWord
197 
198         }
199 
200 
201     }
202 
203     /**
204      * Try to go in one direction. If there is an exit, enter the new
205      * room, otherwise print an error message.
206      */
207     private void goRoom(Command command)
208     {
209         if(!command.hasSecondWord()) {
210             // if there is no second word, we don't know where to go...
211             System.out.println("Go where?");
212             return;
213         }
214 
215         boolean success = true;
216         String direction = command.getSecondWord();
217 
218         // Try to leave current room.
219         Door nextRoomDoor = currentRoom.getDoor(direction);
220 
221         if (nextRoomDoor == null || !nextRoomDoor.hasDoor()) {
222             System.out.println("Are you tryin' to open a door that doesn't
                    exist? That might be hard.");
223             success=false;
224         }
225         else if(nextRoomDoor.hasPassword())
226         {
227             System.out.println("The room is locked. Enter the password.
                    Use the 'unlock' command together with the password");
228             Command pass = parser.getCommand();
229             if(nextRoomDoor.doUnlock(pass.getSecondWord()))
230             {
231                 System.out.println("Success. The door is unlocked.");
232                 success =true;
233             }
234             else
235             {
236                 System.out.println("Failure. The door is still
                        unclocked.");
237                 success=false;
238             }
239 
240         }
241 
242         if(success)
243         {
244             currentRoom = nextRoomDoor.getRoom();
245             System.out.println(currentRoom.getLongDescription());
246         }
```

```
247     }
248
249     /**
250      * "Quit" was entered. Check the rest of the command to see
251      * whether we really quit the game.
252      * @return true, if this command quits the game, false otherwise.
253      */
254     private boolean quit(Command command)
255     {
256         if(command.hasSecondWord()) {
257             System.out.println("Quit what?");
258             return false;
259         }
260         else {
261             return true;  // signal that we want to quit
262         }
263     }
264
265     private String JockumResponse(String input)
266     {
267         input = input.toLowerCase();
268         if(input.contains("leibniz"))
269         {
270             return "Of course! His notation is still used today. The key
                        to the E-Building is 'komm14'. Good luck";
271         }
272         else if (input.contains("hi"))
273         {
274             return "Hi! What can I help you with?";
275         }
276         else if(input.contains("newton"))
277         {
278             return "Well, ....";
279         }
280         else if(input.contains("e") && input.contains("building"))
281         {
282             return "You must have a key to enter the E building. You must
                        answer one of my questions. \n Q: Who made the largest
                        contribution to Calculus? Leibniz or Newton?";
283         }
284         else
285         {
286             return "I don't understand your question.";
287         }
288
289     }
290
291     private void TransformUsers()
292     {
293         Random rn = new Random();
294
295         if(!rn.nextBoolean())
296         {
297             Person temp = E.getPerson();
298             E.setPerson(temp);
299             outside.setPerson(temp);
```

```
300         }
301     }
302
303     private String JonasResponse(String input)
304     {
305         if(!currentRoom.equals(outside))
306         {
307             if(tweetCount  == 20)
308             {
309                 E.setExit("east", null);
310                 return "Hi! The course has now started and all doors are
                        locked. In order to pass the course, 20 tweets have to
                        be written. Use the 'tweet' command to perform this
                        task. When done, tell me.";
311             }
312             else if (tweetCount > 0)
313             {
314                 return "You have " + tweetCount + " left!";
315             }
316             else
317             {
318                 E.setExit("east", outside);
319                 E.setPerson(null);
320                 return "Great, you've passed the course! Thank you for
                        being such a benevolent person by contributing to
                        research! Type 'east' to exit the room";
321             }
322         }
323         else
324         {
325             return "We have a lecture today in E. You can either wait or
                    ask Jockum about the key to the door if you want to get in
                    earlier.";
326         }
327
328     }
329 }
```

```
1
2 public class Launcher {
3
4     public static void main(String[] args) {
5         // TODO Auto-generated method stub
6         Game gm = new Game();
7         gm.play();
8     }
9
10 }
```

```
1 import java.util.Scanner;
2
3 /**
4  * This class is part of the "World of Zuul" application.
5  * "World of Zuul" is a very simple, text based adventure game.
6  *
7  * This parser reads user input and tries to interpret it as an "Adventure"
```

```
8   * command. Every time it is called it reads a line from the terminal and
9   * tries to interpret the line as a two-word command. It returns the
        command
10  * as an object of class Command.
11  *
12  * The parser has a set of known command words. It checks user input
        against
13  * the known commands, and if the input is not one of the known commands,
        it
14  * returns a command object that is marked as an unknown command.
15  *
16  * @author  Michael Ã¶Kling and David J. Barnes
17  * @version 2011.08.10
18  */
19 public class Parser
20 {
21     private CommandWords commands;  // holds all valid command words
22     private Scanner reader;         // source of command input
23
24     /**
25      * Create a parser to read from the terminal window.
26      */
27     public Parser()
28     {
29         commands = new CommandWords();
30         reader = new Scanner(System.in);
31     }
32
33     /**
34      * @return The next command from the user.
35      */
36     public Command getCommand()
37     {
38         String inputLine;   // will hold the full input line
39         String word1 = null;
40         String word2 = "";
41
42         System.out.print("> ");      // print prompt
43
44         inputLine = reader.nextLine();
45
46         // Find up to two words on the line.
47         Scanner tokenizer = new Scanner(inputLine);
48         if(tokenizer.hasNext()) {
49             word1 = tokenizer.next();       // get first word
50             while(tokenizer.hasNext()) {
51                 word2 += tokenizer.next();       // get second word
52                 // note: we just ignore the rest of the input line.
53             }
54         }
55         tokenizer.close();
56         return new Command(commands.getCommandWord(word1), word2);
57     }
58
59     /**
```

```
60          * Print out a list of valid command words.
61          */
62         public void showCommands()
63         {
64             commands.showAll();
65         }
66     }
```

```
1   import java.util.function.Function;
2
3   public class Person {
4       private String name;
5
6       private Function <String, String> responses;
7
8       public Person(String name, Function<String,String> respons)
9       {
10          this.name=name;
11          responses = respons;
12      }
13      public void addResponse(Function <String, String> response)
14      {
15          //this.responses.add(response);
16          this.responses = response;
17      }
18
19      public String getName()
20      {
21          return name;
22      }
23
24      public String getResponse(String message)
25      {
26          return this.responses.apply(message);
27      }
28
29  }
```

```
1
2   public class Quest {
3
4   }
```

```
1   import java.util.Set;
2   import java.util.HashMap;
3
4   /**
5    * Class Room - a room in an adventure game.
6    *
7    * This class is part of the "World of Zuul" application.
8    * "World of Zuul" is a very simple, text based adventure game.
9    *
10   * A "Room" represents one location in the scenery of the game.  It is
11   * connected to other rooms via exits.  For each existing exit, the room
12   * stores a reference to the neighboring room.
13   *
```

```
14  * @author  Michael Ã¶Klling and David J. Barnes
15  * @version 2011.08.10
16  */
17
18 public class Room
19 {
20     private String description;
21     private HashMap<String, Door> exits;        // stores exits of this
        room.
22
23     private Person person;
24
25     /**
26      * Create a room described "description". Initially, it has
27      * no exits. "description" is something like "a kitchen" or
28      * "an open court yard".
29      * @param description The room's description.
30      */
31     public Room(String description)
32     {
33         this.description = description;
34         exits = new HashMap<String, Door>();
35     }
36
37     public void setExit(String direction, Room neighbor)
38     {
39         setExit(direction, neighbor, "");
40     }
41     /**
42      * Define an exit from this room.
43      * @param direction The direction of the exit.
44      * @param neighbor  The room to which the exit leads.
45      */
46     public void setExit(String direction, Room neighbor, String password)
47     {
48         exits.put(direction, new Door(neighbor, password));
49     }
50
51     /**
52      * @return The short description of the room
53      * (the one that was defined in the constructor).
54      */
55     public String getShortDescription()
56     {
57         return description;
58     }
59
60     /**
61      * Return a description of the room in the form:
62      *     You are in the kitchen.
63      *     Exits: north west
64      * @return A long description of this room
65      */
66     public String getLongDescription()
67     {
68         String longDes = "You are " + description + ".\n " ;
```

```
69
70          if(person != null)
71          {
72              longDes += person.getName() + " is in the room. To ask, use
                    'ask' command followed by the question. \n";
73          }
74
75          longDes +=  getExitString();
76
77          return longDes;
78      }
79      /**
80       * Return a string describing the room's exits, for example
81       * "Exits: north west".
82       * @return Details of the room's exits.
83       */
84      private String getExitString()
85      {
86          String returnString = "Exits:";
87          Set<String> keys = exits.keySet();
88          for(String exit : keys) {
89              returnString += " " + exit;
90          }
91          return returnString;
92      }
93
94      /**
95       * Return the room that is reached if we go from this room in direction
96       * "direction". If there is no room in that direction, return null.
97       * @param direction The exit's direction.
98       * @return The room in the given direction.
99       */
100     public Room getExit(String direction)
101     {
102         return exits.get(direction).getRoom();
103     }
104
105     public Door getDoor(String direction)
106     {
107         return exits.get(direction);
108     }
109
110     public void setPerson(Person person)
111     {
112         this.person = person;
113     }
114
115     public Person getPerson()
116     {
117         return person;
118     }
119 }
```