|  |
| --- |
| **Exercise 8.1**  a) What does this application do? "World of Zuul" is a very simple, text based adventure game. Users can walk around some scenery. That's all.  b) What commands does the game accept? help quit go "somewhere"  c) What does each command do? help: Gives information about the commands available quit: Exits the game go "somewhere": Goes through the door in the specified direction. Directions can be one of these: north, east, south, and west.  d) How many rooms are in the scenario? There are 5 rooms.  e) Draw a map of the existing rooms. |
| **Exercise 8.2**   The descriptions below are taken from the documentation in the source code of the classes  **Parser:** This parser reads user input and tries to interpret it as an "Adventure" command. Every time it is called it reads a line from the terminal and tries to interpret the line as a two-word command. It returns the command as an object of class Command. The parser has a set of known command words. It checks user input against the known commands, and if the input is not one of the known commands, it returns a command object that is marked as an unknown command.  **Game:** This class is the main class of the "World of Zuul" application.  To play this game, create an instance of this class and call the "play" method. This main class creates and initializes all the others: it creates all rooms, creates the parser and starts the game. It also evaluates and executes the commands that the parser returns.  **Command:** 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>.  **CommandWords:** This class holds an enumeration of all command words known to the game. It is used to recognize commands as they are typed in.  **Room:** A "Room" represents one location in the scenery of the game. It is connected to other rooms via exits. The exits are labeled north, east, south, west. For each direction, the room stores a reference to the neighboring room, or null if there is no exit in that direction. |
| **Exercise 8.5**  Add the method printLocationInfo() from Code 8.2 to the class Game. Then replace the corresponding lines from the method printWelcome() and goRoom() with a call to the method: printLocationInfo() |
| **Exercise 8.7**  Add this method to the Room class:  /\*\*  \* Return a string describing the room's exits, for example  \* "Exits: north west".  \*/  public String getExitString()  {   String returnString = "Exits: ";  if(northExit != null) {  returnString += "north ";  }  if(eastExit != null) {  returnString += "east ";  }  if(southExit != null) {  returnString += "south ";  }  if(westExit != null) {  returnString += "west ";  }  return returnString;  }  Modify printLocationInfo() in the Game class like this:  private void printLocationInfo()  {  System.out.println("You are " +  currentRoom.getDescription());  System.out.print(currentRoom.getExitString());  System.out.println();  } |
| **Exercise 8.8**  See the *zuul-better* project. |
| **Exercise 8.9**  Taken from the API documentation:  “Returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.” |
| **Exercise 8.10**  First, a string called returnString is created with the initial text "Exits: ". We will then add the exits to this string and finally return it. The names of the available exits are added by retrieving the set of keys from the HashMap of exits. We then iterate through the set of keys and in each iteration we add the key of the exit to returnString. |
| **Exercise 8.11**  See the *zuul-better* project included on the CD. |
| **Exercise 8.12**  The objects are: game1:Game  -->parser:Parser  --> commands:CommandWords  -->outside:Room  -->theatre:Room  -->lab:Room  -->office:Room  -->pub:Room |
| **Exercise 8.13**  The reference from game1:Game to the outside:Room is changed to the new room that we have moved into. If we use the command go east the reference will be to theatre:Room. |
| **Exercise 8.15**  private void eat() {  System.out.println(“You have eaten now and you are not hungry any more”): }  adding:  else if(commandWord.equals("eat")) {  eat(); }  to the command-testing statements. |
| **Exercise 8.17**  No, you don't need to change Game class. The list of commands which are printed out is generated from the array of validCommands, and will therefore automatically include any new commands that have been added to this array. |
| **Exercise 8.19**  **See 08-19-zuul-mvc**  Modify the printHelp() method in Game so the last lines is:  System.out.println(parser.getCommandList());  Add the following method to Parser:  public String getCommandList()  {  return commands.getCommandList();  }  And remove the now obsolete showCommands() from Parser (if you implemented this in Exercise 8.16)  In CommandWords add this method:  public String getCommandList()  {  String commandList = "";  for(String command : validCommands) {  commandList += command + " ";  }  return commandList;  }  And remove the now obsolete showCommands() from CommandWords (if you implemented this in Exercise 8.16) |
| **Exercise 8.22**  Information about the model-view-control (MVC) pattern can be found here:  <https://www.geeksforgeeks.org/mvc-design-pattern/>  A simple example of a Java program that uses MVC:  [http://csis.pace.edu/~bergin/mvc/mvcgui.html](http://csis.pace.edu/%7Ebergin/mvc/mvcgui.html)  For more examples of the MVC pattern look at the Java Swing implementation which makes heavy use of the MVC pattern.  The MVC pattern is related to the discussion in this chapter because it is a pattern that decouples objects into three types of objects: Model objects which represent the data; View objects which handle the display; and Control objects which handle events that modify the View or Model objects. In this chapter we only discussed the separation of View and Model - adding another level of decoupling makes the design even more flexible.  To apply the MVC pattern to the Zuul game, we need to split the application into a model, view and control. We might do something like this:  Model: The Game and Room classes represent the model. We might split the Game class into two classes one which represents the model and one which does the rest. The changes we make to the model while playing the game, is to change the game object's reference to the current room. Whenever we change this reference the model should fire an event to all registered listeners (the View).  View: We should create a new class which handles the view of the model - that is, printing the text to the screen when an update is received from the model.  Controller: As it is now, the control of the game is done from the Game class in the play() and processCommand(Command command) methods.  An example of Zuul with the MVC pattern applied can be downloaded here: 08-19-zuul-mvc.zip |
| **Exercise 8.23**  See Exercise 8.25. (Although the rooms can hold several items) |
| **Exercise 8.24**  a) How should the information about an item present in a room be produced? The items are in the rooms, and hence the room should produce the information about items present.  b) Which class should produce the string describing the item? The Item class should produce the string.  c) Which class should print the description of the item? The Game class is responsible for printing, and hence should also print the description of an item. It is, however, not necessary to explicitly print the item description in the game class if the description of the room includes the description of the item in the room. |
| **Exercise 8.25**  See: 08-25-zuul-with-items.zip |
| **Exercise 8.26**  Add the command "back" to the CommandWords.  The rest of the modifications are in the Game class: Add a field:  private Room previousRoom;  In processCommand() add:  else if (commandWord.equals("back")) {  goBack(command);  }  Introduce a new method enterRoom, which stores the previousRoom. Update the method goRoom() to use this method.  /\*\*  \* Enters the specified room and prints the description.  \*/  private void enterRoom(Room nextRoom)  {  previousRoom = currentRoom;  currentRoom = nextRoom;  System.out.println(currentRoom.getLongDescription());  }  Add this method:  /\*\*  \* Go back to the previous room.  \*/  private void goBack(Command command)  {  if(command.hasSecondWord()) {  System.out.println("Back where?");  }  if (previousRoom == null) {  System.out.println("You have nowhere to go back to!");  }  else {  enterRoom(previousRoom);  }  } |
| **Exercise 8.27**  When a second word is typed after back, it prints an error message: "Go where?"  Another case of *negative testing*: When the game is just started, there is no previous room. In the above implementation this is handled by printing a message to the user: "You can't go back to nothing!" |
| **Exercise 8.28**  If back is typed in twice you end up in the same room as where you were when you typed back the first time. Yes, this is sensible, but it might be more useful to be able to go back several steps - see the next exercise. |
| **Exercise 8.29**  See: 08-29-zuul-back.zip |
| **Exercise 8.31**  There are many possible tests for the zuul project. It is important to have both *positive* and *negative tests*.  Some of the tests could be:  - testing that the rooms are properly connected. - testing that all the commands are recognised and works as expected. - testing the back command as explained in the solution to Exercise 8.27 |
| **Exercise 8.32**  See: 08-32-zuul-refactored.zip |
| **Exercise 8.33 - 8.37**  All the modifications suggested in Exercises 8.33 through 8.37 is implemented in this project:  See: 08-37-zuul-with-player.zip |
| **Exercise 8.39**  Add this in Game.processCommand:  else if (commandWord == CommandWord.LOOK) {  look();  }    And add this method to Game:  private void look()  {  System.out.println(currentRoom.longDescription());  }  And finally, modify the CommandWord to include the new value LOOK:  public enum CommandWord  {  // A value for each command word, plus one for unrecognised  // commands.  GO, QUIT, HELP, LOOK, UNKNOWN;  }  Oh, and don't forget to specify the text associated with the command in the CommandWords constructor:  validCommands.put("look", CommandWord.LOOK); |
| **Exercise 8.40**  Using different command words only requires changes in the CommandWords class. |
| **Exercise 8.41**  When the command word for help is changed it is NOT changed in the welcome message. |
| **Exercise 8.43**  public enum Direction {  NORTH, SOUTH, EAST, WEST } |
| **Exercise 8.44**  Almost. You also need to add the functionality of it to the Game class. Compared to Exercise 8.39 you have one less class to modify in this exercise. |
| **Exercise 8.45**  Yes. It is just using the enum itself: CommandWord.HELP. This will return the command string because toString() has been overridden in CommandWord. |
| **Exercise 8.46**  See: 08-46-zuul-with-timelimit.zip |
| **Exercise 8.48**  To implement a trapdoor (one way door), simply remove one of the exits. For instance, you could remove the exit from the pub to the outside by removing this line:  pub.setExit("east", outside); |
| **Exercise 8.49**  See: 08-49-zuul-with-beamer.zip |
| **Exercise 8.50**  See: 08-50-zuul-with-doors.zip |
| **Exercise 8.51**  See: 08-51-zuul-with-transporter.zip |
| **Exercise 8.52**  See: 08-52-zuul-even-better.zip |
| **Exercise 8.55**  The main method could look like this:  public static void main(String args[])  {  Game game = new Game();   game.play();  } |