Exercise 7.1

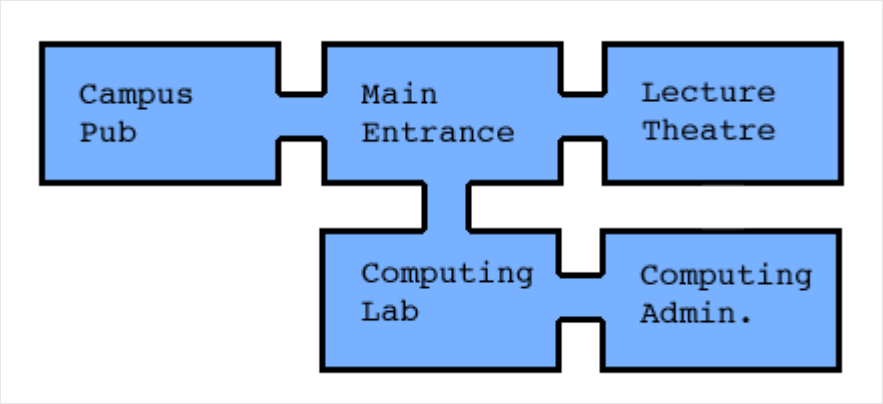
a) What does this application do?
"World of Zuul" is a very simple, text based adventure game. Users can walk around some scenery. That's all.

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 7.2

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 initialises 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 recognise 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 neighbouring room, or null if there is no exit in that direction.

Exercise 7.5

Add the method *printLocationInfo()* from Code 7.2 to the class Game. Then replace the corresponding lines from the method *printWelcome()* and *goRoom()* with a call to the method: *printLocationInfo()*

Exercise 7.7

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 the 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 7.8

See the zuul-better project included on the CD.

Exercise 7.9

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. The set supports element removal, which removes the corresponding mapping from this map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.

Exercise 7.10

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 the returnString.

Exercise 7.11

See the zuul-better project included on the CD.

Exercise 7.12

```
The objects are:
game1:Game
 -->parser:Parser
     -->  commands:CommandWords
 -->outside:Room
     -->theatre:Room
     -->lab:Room
     -->office:Room
     -->pub:Room
```

Exercise 7.13

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 7.14

See page 221 for the implementation details.

Exercise 7.15

Similar to 7.14.

Exercise 7.17

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 7.18

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 7.16)

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 7.16)

Exercise 7.19

Information about the model-view-control (MVC) pattern can be found here:

http://www.enode.com/x/markup/tutorial/mvc.html

A simple example of a Java program that uses MVC:

http://csis.pace.edu/~bergin/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: 07-19-zuul-mvc.zip

Exercise 7.20

See exercise 7.22. (Although the rooms can hold several items)

Exercise 7.21

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 7.22

Download: 07-22-zuul-with-items.zip

Exercise 7.23

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?");
            return;
        }
        if (previousRoom == null) {
            System.out.println("You have nowhere to go back to!");
        }
        else {
            enterRoom(previousRoom);
        }
    }
```

Exercise 7.24

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 7.25

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 7.26

Download: 07-26-zuul-back.zip

Exercise 7.27

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 exercise 7.24

Exercise 7.28

To automate tests for programs that use interactive input there are two options:
- use some kind of input capturing program, that can reproduce the input.
- design your program so it is possible to take the input from another source.

If the last option is chosen, it would be possible to create a test class that reads the input from a text file and verifies the results. To make this kind of change to the zuul project, we would need to modify the Game class to use another parser, and write the new parser that takes the input from a text file rather that from the keyboard. Furthermore, we would need to check the result of each command which would require testing the state of the game (current room, items in the rooms, room history, etc..)

There is an exercise in chapter 12 that explores these ideas.

Exercise 7.29

Download: 07-29-zuul-refactored.zip

Exercise 7.30 - 7.34

All the modifications suggested in exercises 7.30 through 7.34 is implemented in this project:

Download: 07-34-zuul-with-player.zip

Exercise 7.36

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 7.37

Using different command words only requires changes in the CommandWords class.

Exercise 7.38

When the command for help is changed it is NOT changed in the welcome message.

Exercise 7.39

```
public enum Position
{
    TOP, MIDDLE, BOTTOM
}
```

Exercise 7.40

Almost. You also need to add the functionality of it to the game class. Compared to

exercise 7.36 you have one less class to modify in this exercise.

Exercise 7.41

Yes. It is just using the enum itself: CommandWord.HELP. This will return the command string because toString() have been overridden in CommandWord.

Exercise 7.42

Download: 07-42-zuul-with-timelimit.zip

Exercise 7.43

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 7.44

Download: 07-44-zuul-with-beamer.zip

Exercise 7.45

Download: 07-45-zuul-with-doors.zip

Exercise 7.46

Download: 07-46-zuul-with-transporter.zip

Exercise 7.47

Download: 07-47-zuul-even-better.zip

Exercise 7.50

The method signature is: `static int max(int a, int b);`

Exercise 7.51

The methods in the Math class are static because they implement mathematical *function* operations – their results do not depend on an object's state and they always return the same results given the same arguments. Therefore we do not need an object with state to use them. It is also more convenient that you do not have to create an object before calling the method.

Yes, they could have been instance methods, but that would require that you create an instance of the Math class before you could use the methods. The object would have no useful mutable state, only methods.

Exercise 7.52

```
        public static long testLoopTime()
        {
            long startTime = System.currentTimeMillis();
            for(int i = 1; i <= 100; i++) {
                // Do nothing!
            }
            long endTime = System.currentTimeMillis();
            return endTime - startTime;
        }
```

Exercise 7.53

The main method could look like this:

```
    public static void main(String args[])
    {
        Game game = new Game();
        game.play();
    }
```

Exercise 7.56

a) Yes, you can call a static method from an instance method.

b) No, you can not call an instance method from a static method (at least not without first creating an object to call it on).

c) Yes, you can call a static method from a static method.

Exercise 7.57

Yes, you can use a static field and the constructor(s) of the class to count the number of instantiations. If you have more that one constructor, you would need to increase the count in each of the constructors. This is one way to do it:

```
public class Test
{
    private static int instanceCount = 0;

    public Test()
    {
        instanceCount++;
    }

    public Test(String something)
    {
        instanceCount++;
    }

    public static int numberOfInstances()
    {
        return instanceCount;
    }
}
```

It is actually possible to avoid the incrementation in each constructor. You can use an *initialiser block* which is invoked before the constructor call. This is not a structure that is used very often, and you might be best off without telling your students about it. But if someone should ask you about it, here is how it looks:

```
public class Test
{
    private static int instanceCount = 0;

    {
        instanceCount++;
    }

    public Test()
    {
    }

    public Test(String something)
    {
    }

    public static int numberOfInstances()
    {
        return instanceCount;
    }
}
```