

Exercise 6.1

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

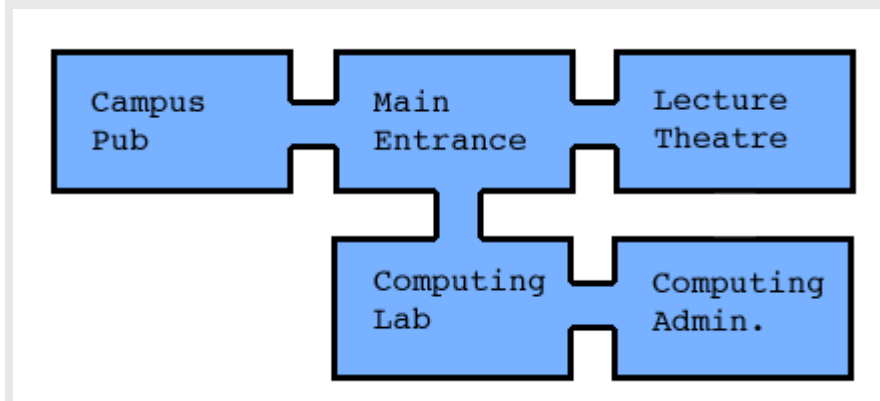
quit: Exits the game

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 6.2

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 6.5

Add the method `printLocationInfo()` from Code 6.2 to the class `Game`. Then replace the corresponding lines from the method `printWelcome()` and `goRoom()` with a call to the method: `printLocationInfo()`

Exercise 6.7

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 6.8

See the *zuul-better* project included on the CD.

Exercise 6.9

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 6.10

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 6.11

See the *zuul-better* project included on the CD.

Exercise 6.12

The objects are:

```

game1:Game
-->parser:Parser
    -->  commands:CommandWords
-->outside:Room
    -->theatre:Room
    -->lab:Room
    -->office:Room
    -->pub:Room

```

Exercise 6.13

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 6.14

See page 216 of the fifth edition for the implementation details.

Exercise 6.15

```
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 6.17

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 6.18

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 6.16)

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 6.16)

Exercise 6.19

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: 06-19-zuul-mvc.zip

Exercise 6.20

See Exercise 6.22. (Although the rooms can hold several items)

Exercise 6.21

- 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 6.22

Download: `06-22-zuul-with-items.zip`

Exercise 6.23

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 6.24

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 6.25

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 6.26

Download: [06-26-zuul-back.zip](#)

Exercise 6.27

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 the solution to Exercise 6.24

Exercise 6.28

Download: [06-28-zuul-refactored.zip](#)

Exercise 6.29 - 6.33

All the modifications suggested in Exercises 6.29 through 6.33 is implemented in this project:

Download: `06-33-zuul-with-player.zip`

Exercise 6.35

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 6.36

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

Exercise 6.37

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

Exercise 6.38

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


Exercise 6.39

Almost. You also need to add the functionality of it to the `Game` class. Compared to Exercise 6.35 you have one less class to modify in this exercise.

Exercise 6.40

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

Exercise 6.41

Download: [06-41-zuul-with-timelimit.zip](#)

Exercise 6.42

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 6.43

Download: [06-43-zuul-with-beamer.zip](#)

Exercise 6.44

Download: [06-44-zuul-with-doors.zip](#)

Exercise 6.45

Download: [06-45-zuul-with-transporter.zip](#)

Exercise 6.46

Download: [06-46-zuul-even-better.zip](#)

Exercise 6.49

The method signature is:

```
static int max(int a, int b);
```

Exercise 6.50

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 6.51

```
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 6.52

The main method could look like this:

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

    game.play();
}
```

Exercise 6.54

The main method could look like this:

```
public static void main(String args[])
{
    SupportSystem system = new SupportSystem();

    system.start();
}
```

Exercise 6.55

- a) Yes, you can call a static method from an instance method.
- b) No, you cannot 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 6.56

Yes, you can use a static field and the constructor(s) of the class to count the number of instantiations. If you have more than 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
    }
}

```