Pavel Tsvyatkov • Stepan Burlachenko
04 | 25 | 2019 • WH C 579

10 | World of You

Index

Assignment

Task 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.

Before we started refactoring we discussed what we were doing in the lectures and had our notes ready so we can move forward to start decoupling the classes. As we were progressing we decided to stop and make sure we are applying correct changes. We ran our game and noticed that we did something wrong, because the game wasn't displaying the possible exits and it was always saying "There is no door". We remembered that the same thing happened in class when we were changing code and had to look thoroughly what was missing, but decided to start from scratch due to the lack of time.

First place where we had code duplication was in the Game class: both the goRoom and the printWelcome methods had lines which printed out pretty much the same message. Instead, we created a new private method in the Game class called printLocationInfo which includes a description of the current location as well as available exists.

Moving to the Room (later Location) code where we need to make sure there potentially can be more than 4 exists. Because we knew from the lecture, that we would have to eventually have all the exits saved as a HashMap, we had to perform encapsulation to reduce coupling: making the exit fields in the Room (later Location) class all private. To implement this, we introduced another method called getExit which is a modifier and can be later used in the Game class to get the private fields of the room.

```
public Room getExit(String direction)
{
 if(direction.equals("north")) {
     return northExit;
    }
 if(direction.equals("east")) {
     return eastExit;
    }
 if(direction.equals("south")) {
     return southExit;
    }
 if(direction.equals("west")) {
         return westExit;
    }
  return null; }
```

Now the goRoom class can be changed to include the getString method, thus making it in no way to depend on the number of possible exists in the Room (later Location) class.

The final change here had to do with making a HashSet of exits and implementing it into the multiple methods of the Room (later Location) class.

First, the setExit method for setting exists for each room. Here, for each room we have a HashMap of direction mapped to the neighbor. The put method of the HashMap class is used to do this.

```
/**
 * Here we can set an exit from this Location.
 * @param direction The exit's direction.
 * @param neighbor  The location to which the exit leads.
 */
public void setExit(String direction, Location neighbor)
{
    exits.put(direction, neighbor);
}
```

Following this change, we modified the createRooms class to use this public method during the Game initialization.

Next change: the getExitString method which uses a Set to get the exits: it gets the names of the exits out of HashMap and concatenates them to a single string.

```
private String getExitString()
{
    String returnString = "Exits:";
    Set<String> keys = exits.keySet();
    for(String exit : keys) {
        returnString += " " + exit;
    }
    return returnString;
}
```

And the very last change is creating a new method which combines the information about the current room description and the available exits. It is called getLongDescription and uses the getExitString to get the available exits.

```
public String getLongDescription()
{
    return "You are " + description + ".\n" + getExitString();
}
```

We don't need the printLocationInfo method anymore and can use the public method of the Room (later Location) class for this.

```
private void printWelcome()
{
    System.out.println();
    System.out.println("Welcome to the Jungles!");
    System.out.println("Your ship has wrecked, you are alone on
    System.out.println("Type 'help' if you need help.");
    System.out.println();
    System.out.println(currentLocation.getLongDescription());
}
```

```
private void goLocation(Command command)
{
    if(!command.hasSecondWord()) {
        // if there is no second word, we don't know where to go...
        System.out.println("Go where?");
        return;
    }

    String direction = command.getSecondWord();

    // Try to leave current room.
    Location nextLocation = currentLocation.getExit(direction);

    if (nextLocation == null) {
        System.out.println("There is no way!");
    }
    else {
        currentLocation = nextLocation;
        System.out.println(currentLocation.getLongDescription())
    }
}
```

The next part of the task was adapting the Zuul to our own game. We changed the field in our Game class from Room to Location to fit to our game and applied changes where necessary so that it's a Location instead of Room.

```
public class Game
{
    private Parser parser;
    private Location currentLocation;
```

We had to change the variable names in our createLocations method, so we added the following locations which we are going to use for our game. We added the corresponding descriptions to them and then applied changes to the possible exits to match our game.

```
/**
 * Create all the locations and link their exits together.
 */
private void createLocations()
{
    Location beach, jungles, temple, shipwreck, volcano, bunker, waterfall;

    // create the loctions
    beach = new Location("on the beach");
    jungles = new Location("deep into the jungles");
    temple = new Location("in an abandoned temple");
    shipwreck = new Location("near the shipwreck");
    volcano = new Location("near the volcano");
    waterfall = new Location("at the waterfall");
    bunker = new Location("near the bunker");

    // initialise location exits
    beach.setExit("east", shipwreck);
    beach.setExit("south", jungles);

    jungles.setExit("north", beach);
    jungles.setExit("east", waterfall);
    jungles.setExit("west", temple);
    jungles.setExit("south", volcano);

    temple.setExit("east", jungles);
    temple.setExit("south", bunker);

    bunker.setExit("north", temple);

    volcano.setExit("north", jungles);

    waterfall.setExit("west", jungles);

    shipwreck.setExit("west", beach);

    currentLocation = beach;  // start game at the beach
}
```

We changed the welcome message and adapted it to our game as well.

```
/**
 * Print out the opening message for the player.
 */
private void printWelcome()
{
    System.out.println();
    System.out.println("Welcome to the Jungles!");
    System.out.println("Your ship has wrecked, you are alone on this island.");
    System.out.println("Type 'help' if you need help.");
    System.out.println();
    System.out.println(currentLocation.getLongDescription());
}
```

We made sure to change the help message to display information about what we need to do.

```
private void printHelp()
{
    System.out.println("You are alone. You need to escape.");
    System.out.println("Find a way to build a boat and get away.");
    System.out.println();
    System.out.println("Your command words are:");
    System.out.println("   go quit help");
}
```

Also, instead of "There is no door" now we were printing "There is no way".

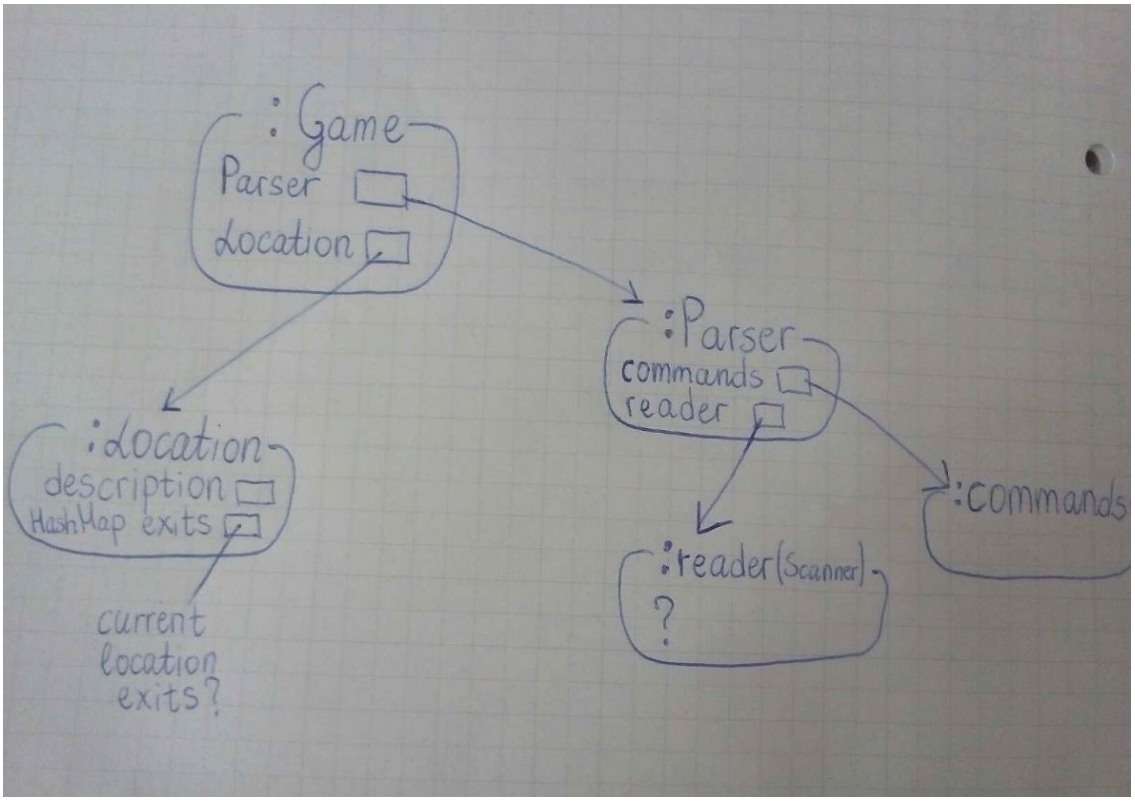
```
if (nextLocation == null) {
    System.out.println("There is no way!");
```

[40 min]

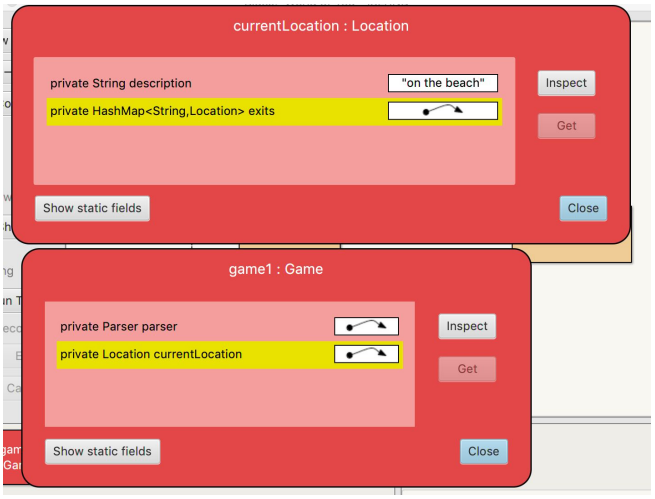
Task 2

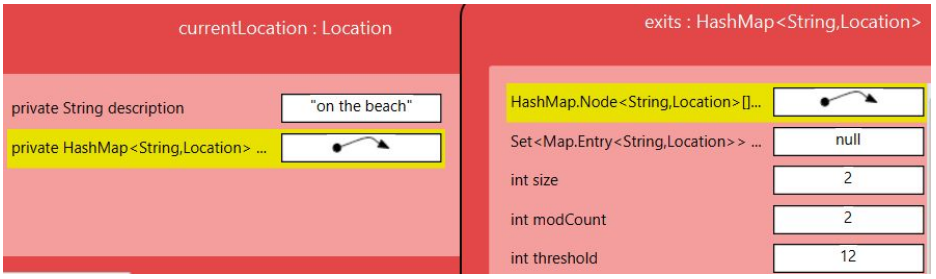
Draw an object diagram showing the state of your system just after it has been started. Does it change if you issue a "go" command?

We inspected the Game object just after we started a new game and tried to sketch an object diagram.



When a new Game object is created, inspecting it shows the current location as the starting point of the game: “on the beach” in our case, while the exits HashMap shows the available exits.

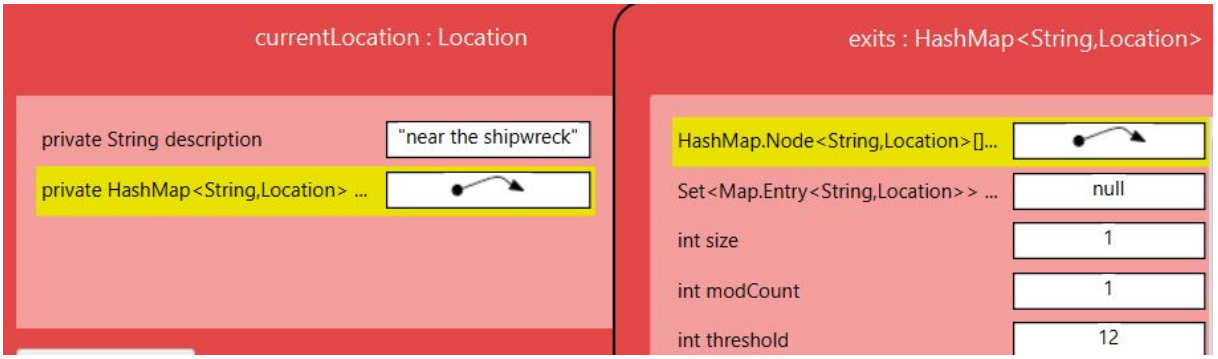




At the beginning, we see "on the beach" as description and the HashMap has a size of 2. We thought that’s because we have 2 available exits: east and south.

We weren’t sure if the object diagram itself changes after issuing a "go" command, but we tested it and saw that there is something that changes.

If we "go east" in our game and inspect the Location, we now see that it changes to “near the shipwreck”. When we go and look inside the exits HashMap, we see it has a size of 1, because there is 1 available exit to the west(going back).



Even though we observed some changes, we were still not sure if the "object diagram" changes. In our opinion it was just the different fields that the objects have changing, but the object diagram was staying the same.

[15 min]

Task 3

Add a "look" command to your game.

To add a new command to the game, changes in two classes had to be made. First, the CommandWords where another valid command has to be added to the validCommands list.

```
public class CommandWords
{
    // a constant array that holds all valid command words
    private static final String[] validCommands = {
        "go", "quit", "help", "look"
    };
```

The next change has to be done in the Game class.

```
/**
 * Listing all the items that can be picked in the current location.
 */
private void listItems()
{
System.out.println("There are no items to pick here");
}
```

We added a method "listItems", which just prints a message for now.

Then we added another else if statement for the look command to the processCommand method, so we can execute it while playing.

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

    if(command.isUnknown()) {
        System.out.println("I don't know what you mean...");
        return false;
    }

    String commandWord = command.getCommandWord();
    if (commandWord.equals("help")) {
        printHelp();
    }
    else if (commandWord.equals("go")) {
        goLocation(command);
    }
    else if (commandWord.equals("quit")) {
        wantToQuit = quit(command);
    }
    else if (commandWord.equals("look")){
        listItems();
    }
```

Lastly, the printHelp method of the Game class needed a little modification as well: the look command had to be added to the list of possible command words there.

```
/**
 * Print out some help information.
 * Here we print some stupid, cryptic message and a list of the
 * command words.
 */
private void printHelp()
{
    System.out.println("You are lost. You need to escape");
    System.out.println("Find a way to build a boat and get away");
    System.out.println();
    System.out.println("Your command words are:");
    System.out.println("   go quit help look");
}
```

We tested it in our game and it worked correctly.

```
You are on the beach.
Exits: east south
> look
There are no items to pick here.
```

[20 min]

Task 4

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.

Just like in the previous exercise we added another command "eat" to the list of validCommands in the CommandWords class.

```
public class CommandWords
{
    // a constant array that holds all valid command words
    private static final String[] validCommands = {
        "go", "quit", "help", "look", "eat"
    };
```

Then we went on and applied the same changes to the Game class. First, we added a new "eat" method, which just displays a message for now.

```
/**
 * For now we just print a message, later we will be able to
 * eat only when we have found food.
 */
private void eat(){
  System.out.println("You have eaten now and aren't hungry any more.");
}
```

Then we added another else if statement to the processCommand method in the Game class, so that we can use our "eat" command in the game.

```
else if (commandWord.equals("eat")){
    eat();
}
```

We added the command to the printHelp method so it gets displayed when we type "help".

```
System.out.println("   go quit help look eat");
```

We knew that this command will need some changes in the next version, because we will also have a second word to the eat command to say what we want to eat, but that's going to happen at a later point when we have added items to our game.

```
Welcome to the Jungles!
Your ship has wrecked, you are alone on this island.
Type 'help' if you need help.

You are on the beach.
Exits: east south
> eat
You have eaten now and aren't hungry any more.
```

We tested it and it was working well.

Task 5

Implement an improved version of printing out the command words.

We noticed that even if we were able to use a command, when we typed help in game, the new commands we added were not displayed, unless we added them manually to the printHelp method. Our printHelp method at this point was looking like that and it needed some changes so we can display the command words in a better way.

```
private void printHelp()
{
    System.out.println("You are alone. You need to escape.");
    System.out.println("Find a way to build a boat and get away.");
    System.out.println();
    System.out.println("Your command words are:");
    System.out.println("   go quit help look eat");
}
```

We thought about it and agreed that if we just add the commands manually to the String, every time we want to make a change we would have to manually edit this method. To print all the commands in a better way, we had to implement some changes to the CommandWords and Parser class.

In the CommandWords class we added the following method, which is going through the list of validCommands and prints them out.

```
public void showAll()
{
    for(String command: validCommands) {
        System.out.print(command + "  ");
    }
    System.out.println();
}
```

Then we went to the Parser class and also added another method showCommands.

```
public void showCommands()
{
    commands.showAll();
}
```

Now with this method in our Parser class, we call the showAll method from the CommandWords class to show all the commands that we have in the validCommands list. In this way, we now only need to make changes in one class when we want to add or remove a command.

Now the last change that had to be made was in the printHelp method in the Game class. Instead of displaying the commands as a String we now call the showCommands method.

```
private void printHelp()
{
    System.out.println("You are alone. You need to escape.");
    System.out.println("Find a way to build a boat and get away.");
    System.out.println();
    System.out.println("Your command words are:");
    parser.showCommands();
}
```

```
You are on the beach.
Exits: east south
> help
You are alone. You need to escape.
Find a way to build a boat and get away.

Your command words are:
go  quit  help  look  eat
```

We tested the help in our game and it worked well.

[20 min]

Task 6

Add another command that fits your game—did you have to change the Game class? Why or why not?

Another command we decided to add was “Pick”, which later can be used to pick an item from the location if it is there and can be useful later.
Just like in Task 3, to add a new command to the game, changes in two classes had to be made. In the CommandWords we added another command to the validCommands list.

```
public class CommandWords
{
    // a constant array that holds all valid command words
    private static final String[] validCommands = {
        "go", "quit", "help", "look", "eat", "pick"
    };
```

Then, the Game class has to include a new method for this new command. Because we suppose that eventually we’ll have some items to pick, we decided to incorporate the same

idea that the goLocation method has: the command should have a second word after ‘pick’ to be performed. Here, we use the same method of the Command class.

```
/**
 * Picking an item
 */
private void pickItem(Command command)
{
 if(!command.hasSecondWord()) {
        // if there is no second word, we don't know what to pick...
        System.out.println("Pick what?");
        return;
    }
}
```

Testing the new method resulted in success.

```
Welcome to the Jungles!
Your ship has wrecked, you are alone on this island.
Type 'help' if you need help.

You are on the beach.
Exits: east south
> help
You are alone. You need to escape.
Find a way to build a boat and get away.

Your command words are:
go  quit  help  look  eat  pick
> pick
Pick what?
```

[10 min]

Evaluation

Pavel

While working on the exercises in this lab I learned more about the importance of loose and tight coupling and why it’s always better to let one unit take care of one thing. I learned how to add new commands to the game and adapt them to our version of the game. It was also interesting to draw an object diagram and see the state of our game in that way. I learned how we can improve some of our methods, so if we need to change them later, we need to make changes in one place only.

Stepan

This Lab has been the most interesting one so far because although we had a pre-made project to work with as opposed to starting a brand new one, it was nice to implement your own ideas into it and see how it works out.
This lab taught me about working with classes as well as to think about the best ways to reduce any tight coupling to make classes as agile as possible.

Appendix

Pre Lab

Pavel

Stepan

P1. Design your own game scenario away from the computer. Don't worry about implementation or classes or programming. Just try and come up with something interesting. It needs to be the basic structure of a player moving through different locations. Possible examples:

- White blood cells traveling through the body in search of viruses to attack
- Finding the exit in a big shopping mall
- A mole must find the food hidden in one of his burrows before winter comes
- An adventurer is looking for a monster in a series of dungeons
- The bomb squad must find the room with the bomb before it goes off.
- The NSA is looking for Ed Snowdon and going from country to country.
- ...

Be creative! Give your game a name.

Surviving a ship-wreck and staying on what seemed to be an uninhabited island at first.

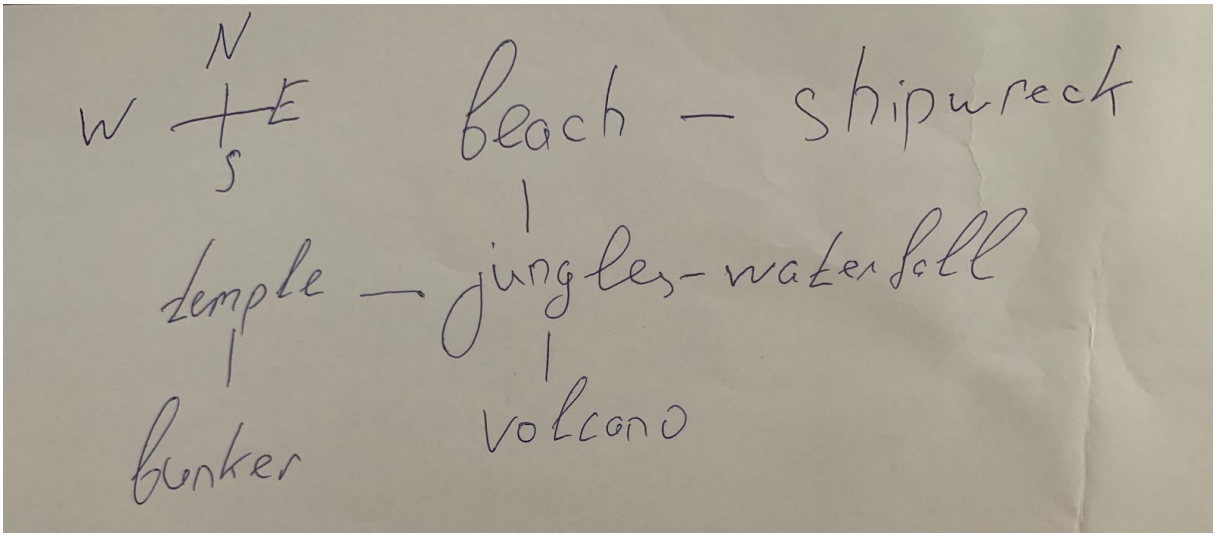
P2. What is the goal of your game, that is, when does the player win?

The goal of the game is to collect all necessary elements around different locations on an island to build a boat.

P3. What could you add to the game to make it interesting? Trap doors, treasure, monsters, ...

The island has different locations, some of them are full of animals, some have people, some have traps.

P4. Draw a map of your game layout.



Final Code

Game class

```
/**
 * World of You - ISLAND
 * You were in a shipwreck. You wake up on an unknown island. You
 * need to find items to rebuild your boat and leave the island.
 * Be careful, noone knows what creatures lurk around the Jungle.
 * @version 2019
 */
public class Game
{
    private Parser parser;
    private Location currentLocation;

    /**
     * Create the game and initialise its internal map.
     */
    public Game()
    {
        createLocations();
        parser = new Parser();
    }

    /**
     * Create all the locations and link their exits together.
     */
    private void createLocations()
    {
        Location beach, jungles, temple, shipwreck, volcano, bunker, waterfall;
```

```
    // create the loctions
    beach = new Location("on the beach");
    jungles = new Location("deep into the jungles");
    temple = new Location("in an abandoned temple");
    shipwreck = new Location("near the shipwreck");
    volcano = new Location("near the volcano");
    waterfall = new Location("at the waterfall");
    bunker = new Location("near the bunker");

    // initialise location exits
    beach.setExit("east", shipwreck);
    beach.setExit("south", jungles);

    jungles.setExit("north", beach);
    jungles.setExit("east", waterfall);
    jungles.setExit("west", temple);
    jungles.setExit("south", volcano);

    temple.setExit("east", jungles);
    temple.setExit("south", bunker);

    bunker.setExit("north", temple);

    volcano.setExit("north", jungles);

    waterfall.setExit("west", jungles);

    shipwreck.setExit("west", beach);

    currentLocation = beach;  // start game at the beach
}

/**
 *  Main play routine.  Loops until end of play.
 */
public void play()
{
    printWelcome();

    // Enter the main command loop.  Here we repeatedly read commands and
    // execute them until the game is over.

    boolean finished = false;
    while (! finished) {
        Command command = parser.getCommand();
        finished = processCommand(command);
    }
    System.out.println("Thank you for playing.  Good bye.");
}
```

```
/**
 * Print out the opening message for the player.
 */
private void printWelcome()
{
    System.out.println();
    System.out.println("Welcome to the Jungles!");
    System.out.println("Your ship has wrecked, you are alone on this island.");
    System.out.println("Type 'help' if you need help.");
    System.out.println();
    System.out.println(currentLocation.getLongDescription());
}

/**
 * Given a command, process (that is: execute) the command.
 * @param command The command to be processed.
 * @return true If the command ends the game, false otherwise.
 */
private boolean processCommand(Command command)
{
    boolean wantToQuit = false;

    if(command.isUnknown()) {
        System.out.println("I don't know what you mean...");
        return false;
    }

    String commandWord = command.getCommandWord();
    if (commandWord.equals("help")) {
        printHelp();
    }
    else if (commandWord.equals("go")) {
        goLocation(command);
    }
    else if (commandWord.equals("quit")) {
        wantToQuit = quit(command);
    }
    else if (commandWord.equals("look")){
        listItems();
    }
    else if (commandWord.equals("pick")){
        pickItem(command);
    }
    // else command not recognised.
    return wantToQuit;
}

// implementations of user commands:
```

```
/**
 * Print out some help information.
 * Here we print some message and a list of the
 * command words.
 */
private void printHelp()
{
    System.out.println("You are alone. You need to escape.");
    System.out.println("Find a way to build a boat and get away.");
    System.out.println();
    System.out.println("Your command words are:");
    parser.showCommands();
}

/**
 * Try to go to one direction. If there is an exit, enter the new
 * location, otherwise print an error message.
 */
private void goLocation(Command command)
{
    if(!command.hasSecondWord()) {
        // if there is no second word, we don't know where to go...
        System.out.println("Go where?");
        return;
    }

    String direction = command.getSecondWord();

    // Try to leave current room.
    Location nextLocation = currentLocation.getExit(direction);

    if (nextLocation == null) {
        System.out.println("There is no way!");
    }
    else {
        currentLocation = nextLocation;
        System.out.println(currentLocation.getLongDescription());
    }
}

/**
 * Listing all the items that can be picked in the current location.
 */
private void listItems()
{
    System.out.println("There are no items to pick here.");
}
```

```
    /**
     * Picking an item
     */
    private void pickItem(Command command)
    {
        if(!command.hasSecondWord()) {
            // if there is no second word, we don't know what to pick...
            System.out.println("Pick what?");
            return;
        }
    }
    /**
     * "Quit" was entered. Check the rest of the command to see
     * whether we really quit the game.
     * @return true, if this command quits the game, false otherwise.
     */
    private boolean quit(Command command)
    {
        if(command.hasSecondWord()) {
            System.out.println("Quit what?");
            return false;
        }
        else {
            return true;  // signal that we want to quit
        }
    }
}
```

Location Class

```
import java.util.Set;
import java.util.HashMap;

/**
 * Class Location - a location in an adventure game.
 *
 * @author   Pavel Tsvyatkov & Stepan Burlachenko
 * @version 2019
 */

public class Location
{
    private String description;
    private HashMap<String, Location> exits;// stores exits of this location.

    /**
     * Create a location described "description". Initially, it has
```

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

/**
 * Here we can set an exit from this Location.
 * @param direction The exit's direction.
 * @param neighbor  The location to which the exit leads.
 */
public void setExit(String direction, Location neighbor)
{
    exits.put(direction, neighbor);
}

/**
 * @return The short description of our Location.
 */
public String getShortDescription()
{
    return description;
}

/**
 * Returns a more detailed description of the location.
 * @return Long description of this location.
 */
public String getLongDescription()
{
    return "You are " + description + ".\n" + getExitString();
}

/**
 * This method shows the possible exits in a location.
 * @return location exits.
 */
private String getExitString()
{
    String returnString = "Exits:";
    Set<String> keys = exits.keySet();
    for(String exit : keys) {
        returnString += " " + exit;
    }
    return returnString;
```

```
    }

    /**
     * Returns the Location we went to and if there wasn't anything
     * in the direction we specified, it should return null.
     * @param direction Direction of the exit.
     * @return After we have moved to a direction, return the location.
     */
    public Location getExit(String direction)
    {
        return exits.get(direction);
    }
}
```

Command Class

```
/**
 * This class is part of the "World of You" application.
 * "World of You" 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   Pavel Tsvyatkov & Stepan Burlachenko
 * @version 2019
 */

public class Command
{
    private String 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(String 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 String getCommandWord()
    {
        return commandWord;
    }

    /**
     * @return The second word of this command. Returns null if there was no
     * second word.
     */
    public String getSecondWord()
    {
        return secondWord;
    }

    /**
     * @return true if this command was not understood.
     */
    public boolean isUnknown()
    {
        return (commandWord == null);
    }

    /**
     * @return true if the command has a second word.
     */
    public boolean hasSecondWord()
    {
        return (secondWord != null);
    }
}
```

Parser Class

```
import java.util.Scanner;

/**
 * This class is part of the "World of You" application.
 * "World of You" is a very simple, text based adventure game.
 *
 * 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.
 *
 * @author   Pavel Tsvyatkov & Stepan Burlachenko
 * @version 2019
 */
public class Parser
{
    private CommandWords commands;  // holds all valid command words
    private Scanner reader;         // source of command input

    /**
     * Create a parser to read from the terminal window.
     */
    public Parser()
    {
        commands = new CommandWords();
        reader = new Scanner(System.in);
    }

    /**
     * @return The next command from the user.
     */
    public Command getCommand()
    {
        String inputLine;   // will hold the full input line
        String 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 = tokenizer.next();      // get first word
```

```
            if(tokenizer.hasNext()) {
                word2 = tokenizer.next();      // get second word
                // note: we just ignore the rest of the input line.
            }
        }

        // Now check whether this word is known. If so, create a command
        // with it. If not, create a "null" command (for unknown command).
        if(commands.isCommand(word1)) {
            return new Command(word1, word2);
        }
        else {
            return new Command(null, word2);
        }
    }

    /**
     * Prints out all available commands.
     */
    public void showCommands()
    {
        commands.showAll();
    }
}
```

CommandWords Class

```
/**
 * This class is part of the World of You - Island application.
 *
 * This class holds all command words known to the game.
 * It is used to recognise commands as they are typed in.
 *
 * @author   Pavel Tsvyatkov & Stepan Burlachenko
 * @version 2019
 */

public class CommandWords
{
    // a constant array that holds all valid command words
    private static final String[] validCommands = {
        "go", "quit", "help", "look", "pick"
    };

    /**
     * Constructor - initialise the command words.
     */
```

```
    public CommandWords()
    {
        // nothing to do at the moment...
    }

    /**
     * Check whether a given String is a valid command word.
     * @return true if it is, false if it isn't.
     */
    public boolean isCommand(String aString)
    {
        for(int i = 0; i < validCommands.length; i++) {
            if(validCommands[i].equals(aString))
                return true;
        }
        // if we get here, the string was not found in the commands
        return false;
    }

    /**
     * Display all commands.
     */
    public void showAll()
    {
        for(String command: validCommands) {
            System.out.print(command + "  ");
        }
        System.out.println();
    }
}
```