Pavel Tsvyatkov • Stepan Burlachenko
07 | 02 | 2019 • WH C 579

11 | Keeping Track of Stuff

Index

Assignment

Task 1

Start by exchanging your game and the items specified in the pre-lab from the last exercise with another group. Read through the code and documentation and see if you can understand what the game is about. Write a short description in your report. Is there anything that needs better explanation? Do you understand what the items are to be? Speak with the authors to make sure you understand.

After we exchanged our game we had to wait for a while to receive the game that we should work on. We received only the game project and no items list, so we just opened the game in BlueJ and started reading.

Unfortunately the game was very messy. It had almost no documentation and the comments were not very helpful to understand what was going on. We decided to read through the source code and see if we could get a better understanding.

We started reading and the code seemed similar to what we had worked on up until now. They had 14 rooms, which we thought was too much.

```
private void createRooms()
{
    Room outside, loft, socialHousing, seniorHome, sharedFlat,first,
    second,stairs,family, holidayFlat, apartment, oneRoom, third, fourth;
```

From the rooms that were getting created we understood that the game has something about apartments and it also has four floors. We looked through all the descriptions of the rooms and got a better idea of the whole picture, even though some of the descriptions were not well structured.

Everything seemed to be fine until we saw how the exits are set.

```
//Treppenhaus betreten
outside.setExit("in",stairs);
first.setExit("in",stairs);
second.setExit("in",stairs);
third.setExit("in",stairs);
fourth.setExit("in",stairs);
```

```
//Treppenhaus verlassen
stairs.setExit("exit",outside);
stairs.setExit("exit",first);
stairs.setExit("exit",second);
stairs.setExit("exit",third);
stairs.setExit("exit",fourth);
```

```
fourth.setExit("down",stairs);
stairs.setExit("down",third);
third.setExit("down",stairs);
stairs.setExit("down",second);
second.setExit("down",stairs);
stairs.setExit("down",second);
first.setExit("down",stairs);
stairs.setExit("down",outside);
```

```
// Treppenhaus nach oben
stairs.setExit("up",first);
first.setExit("up",stairs);
stairs.setExit("up",second);
second.setExit("up",stairs);
stairs.setExit("up",third);
third.setExit("up",stairs);
stairs.setExit("up",fourth);
```

At this point we were very confused and decided to play the game to see if it actually worked. As we expected, the game wasn't working well. We could only access "outside", "stairs", "loft", and "fourth floor", because they used the same command keyword to move around the rooms and only the last places were available to the designated keyword. What was interesting is that the welcome message was saying something about horror game and the description of the room we are first in was saying we are a few steps away from our dream home.

```
Welcome to Horror inspection!
A game which is getting on your nerves
Type 'help' if you need help.

You are Just a few steps away from your dream home.
 If you hurry, you'll be on time, which leaves a good
impression by the apartment providers.
```

At this point we really needed help, to understand what was the point of the game so we typed in help. Unfortunately, there wasn't anything we found helpful.

```
Please, enter command word and your direction, you wanna to go
You can choose up,down, left,right,in and exit

Your command words are:

go  quit  help  look  take  drop  pick  info  back
```

We asked the group what's the point of the game and how exactly we win, but they only had an initial idea, that you should sign a rental document first before anyone else does it, which we couldn't implement for now. They gave us a list with items, but they had no idea yet where the items should be placed. We started thinking how to refactor the game so we can then move on to place the items in rooms where we thought it would be suitable.

We decided to leave just the locations that could be accessed by then: "outside", "stairs", "loft", and "fourth floor".

```
// we commented out all exit, because they arent set properly
// for now we are going to use only those rooms
outside.setExit("in",stairs);
stairs.setExit("down",outside);
stairs.setExit("up",fourth);
fourth.setExit("down",stairs);
fourth.setExit("visit",loft);
loft.setExit("leave",fourth); // from leavet to leave
```

In addition, we made some minor changes to the location descriptions as well as the text messages to fit into the modified by us version of game where there are only 4 locations

available. In this way we could focus on the next exercises and the rest of the rooms could be added later easily with proper descriptions.

[60 min]

Task 2

Extend that project so that a room can contain a single item. Items have descriptions and weights. When creating rooms and setting their exits, items for this game should also be created. When a player enters a room, information about an item present in this room should be displayed.

We discussed this task and both agreed to create a new class called Item. First, we had to declare the fields. We declared three fields for the name, description and weight of an item. We set them to public so that other classes can have access to them as well.

```
public class Item
{
    public static String nameOfItem;
    public String descriptionOfItem;
    public int weightOfItem;
```

Then we created the constructor for Item. We used the same names in the parameter list so we knew what we are referring to. Because of this we also had to use the "this" keyword, so that the constructor knows we are referring to the field called "nameOfItem" on the left hand side.

```
/**
 * Constructor for the Item class.
 * @param nameOfItem The name we are giving to an item.
 * @param descriptionOfItem The way we describe the item.
 * @param weightOfItem The weight of an item.
 */
public Item(String nameOfItem,String descriptionOfItem, int weightOfItem)
{
    this.nameOfItem = nameOfItem;
    this.descriptionOfItem = descriptionOfItem;
    this.weightOfItem = weightOfItem;
}
```

After that we created three accessor methods so we can return the name, description and the weight of an item. They all don’t take any parameters.

```
/**
* @return Returns the name of an item.
*/
public static String getNameOfItem()
{
    return nameOfItem;
}
```

```
/**
 * @return Returns the description of an item.
 */
public String getDescriptionOfItem()
{
    return descriptionOfItem;
}
```

```
/**
 * @return Returns the weight of an item.
 */
public int getWeightOfItem()
{
    return weightOfItem;
}
```

After doing this, we thought we are now ready with the new Item class and went on to edit the Room class.

In the Room class we added a method called "addItem" which adds an item.

```
/**
 * Add an item to a room.
 * @param item
 */
public void addItem(Item item){
    item = item;
}
```

Then we moved to the Game class, to implement the next changes. First, when the rooms are created, we also create the items.

```
private void createRooms()
{
    Room outside, loft, stairs, fourth;

    /*socialHousing, seniorHome, sharedFlat,first,
    second, ,family, holidayFlat, apartment, oneRoom,third
    */
    Item item1, item2, item3;
```

In the same method first we create a new Item and pass the needed parameters and then we add the item to the corresponding Room.

```
item1 = new Item("Magazine","A nice magazine with apartments for sale",5);
outside.addItem(item1);



currentRoom = outside;  // start game outside
```

Then we needed to change the printWelcome method so that Items are displayed as well as soon as we start the game. In the displayed message we print "Items" and then call the getNameOfItem method from the Item class.

```
/**
 * Print exit the opening message for the player.
 */
private void printWelcome()
{
    System.out.println("You are just a few steps away from your new home.");
    System.out.println("Go inside and find a nice one.");
    System.out.println("Type 'help' if you need help.");
    System.out.println();
    System.out.println(currentRoom.getLongDescription());
    System.out.println("Items: " + Item.getNameOfItem());
}
```

Another place we had to add the same line was in the goRoom method, so if there was an item in the next room it will also get displayed.

```
private void goRoom(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.
    Room nextRoom=currentRoom.getExit(direction);
    if(nextRoom==null){
        System.out.println("There is no exit!");
    }else{

        currentRoom=nextRoom;

        System.out.println(currentRoom.getLongDescription());
        System.out.println("Items: " + Item.getNameOfItem());

    }
}
```

We went on to play the game and test if we have an Item "outside". As soon as we started the game we saw that Items are getting displayed as well.

```
You are just a few steps away from your new home.
Go inside and find a nice one.
Type 'help' if you need help.

You are outside the residential complex. Will you go in?
Exits:  in
Items: Magazine
>
```

[40 min]

Task 3

How should the information about an item present in a room be produced? Which class should produce the string describing the item? Which class should print it? Why? If answering this exercise makes you feel you should change your implementation, go ahead and do so – and explain in your report why.

We both discussed this task and at first thought it would be best if the Room class has the method to print the item and the Game class should be the one that prints it. We have the method that describes an item in the Item class for now, because we are not calling it from other methods.

We included a "look" method in the Game class that prints the description of the current room and also displays all the Items that are in this room.

```
/**
 * With this method we print both the description of the room and
 * all the items that are stored in the room.
 */
private void look(){
    System.out.println(currentRoom.getLongDescription());
    System.out.println("Items: " + currentRoom.getItemsName());
}
```

The look method is now used by all other methods that used to display the description of the current room.
[15 min]

Task 4

Modify the project so that a room can hold any number of items. Use a collection to do this! Make sure the room has an addItem method that places an item into the room.

Make sure all items get shown when a player enters a room. Set up a fixture to thoroughly test this!

For this exercise we had to switch to an ArrayList for storing all the items available in this or that room. The ArrayList collection is included into the Room class. In the class fields we included a new ArrayList of Items called roomItems while the constructor created a new array list of this kind.

```
public class Room
{
    public String description;
    public HashMap<String, Room> exits;
    public ArrayList<Item> roomItems;
```

```
public Room(String description)
{
    this.description = description;
    exits = new HashMap<>();
    roomItems = new ArrayList<Item>();

}
```

Next, a new method called getItemsName to return a string of all the items in the array list. First it is empty and if there are no items in the room, then it returns an empty string. Then, if it is not empty, there is a for-loop which goes over all the items in a room and the name of each item gets stored in "itemNames".

```
public String getItemsName(){
    String itemNames = "";
    if(roomItems.size() == 0){
        return " ";
    }
    for(Item items : roomItems){
        itemNames = "" + items.getItemName();
    }
    return itemNames;
}
```

In the same Room class, we include a new method to add items to our array list, so that a room can hold more than one item.

```
public void addItem(Item item){
    roomItems.add(item);
}
```

This method is called in the game class to assign items to a particular room. For now we have two items in the room called "fourth" only.

```
outside.addItem(document);
stairs.addItem(card);
fourth.addItem(hat);
fourth.addItem(key);
```

To display the items stored in the array list for each room, we included the look method in the Welcome message and in the goRoom method (as well as the goBack method later). The look method prints out the description of the current room and the array list using the "getItemsName" method of the Room class.

```
private void look(){
    System.out.println(currentRoom.getLongDescription());
    System.out.println("Items: " + currentRoom.getItemsName());
}
```

Now when we move to different rooms, the items in each room get displayed.

```
> You are just a few steps away from your new home.
Go inside and find a nice one.
Type 'help' if you need help.

You are outside the residential complex. Will you go in?.
Exits: in
Items:  Magazine
> go in
You are in the staircase of the residential complex. You can go up from here.
Exits: up down
Items:  card
```

[30 min]

Task 5

Implement a "back" command that does not have a second word and takes the player back into the previous location. Test this! What happens if a second word is given?

When we got the project, there was already a “back” command implemented but it just gave out a message printed to the terminal. We deleted it and got onto creating the correct version for it.

First, we added an additional field of type Room with the name lastRoom.

```
public class Game
{
    public Parser parser;
    public Room currentRoom;
    public Room lastRoom;
```

Since when we received the project there was a switch statement for all the commands, instead of the traditional else if statements for each command, in the case when we call a "go" command we added that the currentRoom should get stored in the lastRoom.

```
switch(commandWord){
    case "go":
    lastRoom = currentRoom;
    goRoom(command);
    break;
```

The next thing we had to do is to add a new method called goBack. In the method's body we now say that the lastRoom gets stored in the variable currentRoom and we print the description of the room.

```
private void goBack(){
 currentRoom = lastRoom;
 System.out.println(currentRoom.getLongDescription());
}
```

Then the last thing to do was to add the goBack method to the case "back" in the switch statement.

```
case "back":
goBack();
break;
```

We tried to play and see if our back command works correctly.

```
You are just a few steps away from your new home.
Go inside and find a nice one.
Type 'help' if you need help.

You are outside the residential complex. Will you go in?.
Exits: in
Items: Magazine
> go in
You are in the staircase of the residential complex. You can go up from here.
Exits: up down
Items: card
> back
You are outside the residential complex. Will you go in?.
Exits: in
Items: Magazine
>
```

We tested the game and the back command seemed to work as we wanted to. Then we tried what happens if we input a second word after the "back" command.

When we input something following the word “back”, the command is still executed.

```
You are outside the residential complex. Will you go in?.
Exits: in
Items: Magazine
> go in
You are in the staircase of the residential complex. You can go up from here.
Exits: up down
Items: card
> back a
You are outside the residential complex. Will you go in?.
Exits: in
Items: Magazine
```

[20 min]

Evaluation

Pavel

Working on this lab was interesting, because after working on our own project we had to swap it with another group and communicate with each other. We had to make sure we understand the code correctly and it helped me to understand the importance of documenting our projects thoroughly. I got a better understanding on how to read other people’s code and ask questions where needed, if something was not clear. We had to include comments on what we changed, so that the other group can see how we worked. I also learned how to add items to the rooms that already exist and it made me wonder about working with the item’s weight next.

Stepan

Although it was interesting to work with someone else's project on the one hand, on the other hand, I always had an urge to change things to my liking even if there was no sufficient reason to do so sometimes (more often than not, there was however). When working on this lab, I once again realised the importance of providing sufficient comments and testing every little thing.

Appendix

Final Code

Game class

```
import java.util.*;
```

```
/**
 *  This class is the main class of the "World of Zuul" application.
 *  "World of Zuul" is a very simple, text based adventure game.  Users
 *  can walk around some scenery. That's all. It should really be extended
 *  to make it more interesting!
 *
 *  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.
 *
 *  @author  Michael Kolling and David J. Barnes
 *         Modified by Hikari Sophia Stölzle and Nsimba Antonio
 *         Modified by Pavel Tsvyatkov and Stepan Burlachenko
 * @version 2019.07.06
 */

public class Game
{
    public Parser parser;
    public Room currentRoom;
    public Room lastRoom;

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

    }

    /**
     * Create all the different Flats  and link their exits together.
     */
    private void createRooms()
    {
        Room outside, loft, stairs, fourth;

        /*socialHousing, seniorHome, sharedFlat,first,
        second, ,family, holidayFlat, apartment, oneRoom,third
        */
        Item item1, item2, item3;

        outside = new Room("outside the residential complex. Will you go in?");
        item1 = new Item("Magazine","A nice magazine with apartments for sale",5);
```

```
        outside.addItem(item1);

        loft = new Room("in the loft. The place has a very nice view.");

        stairs = new Room("in the staircase of the residential complex. You can go up from
here.");
        item2 = new Item("item2", "this is item2", 5);
        stairs.addItem(item2);

        fourth = new Room ("on the fourth floor. There seems to be a nice apartment.");

        // we commented out all exits, because they arent set properly
        // for now we are going to use only those rooms

        outside.setExit("in",stairs);
        stairs.setExit("down",outside);
        stairs.setExit("up",fourth);
        fourth.setExit("down",stairs);
        fourth.setExit("visit",loft);
        loft.setExit("leave",fourth); // from leavet to leave

        currentRoom = outside;  // start game outside
    }

    /**
     *  Main play method.  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 exit the opening message for the player.
     */
    private void printWelcome()
    {
        System.out.println("You are just a few steps away from your new home.");
```

```
    System.out.println("Go inside and find a nice one.");
    System.out.println("Type 'help' if you need help.");
    System.out.println();
    System.out.println(currentRoom.getLongDescription());
    System.out.println("Items: " + Room.getNameOfItem());
}



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

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

        String commandWord = command.getCommandWord();

        switch(commandWord){
            case "go":
            lastRoom = currentRoom;
            goRoom(command);
            break;

            case"help":
            printHelp();
            break;
            case "look":
            System.out.println("Items: "+currentRoom.getNameOfItem());
            break;
            case "back":
            goBack();
            break;
            case "pick":
            System.out.println("There is nothing to pick up yet.");
            break;
            case "eat":
            System.out.println("You found something to eat!");
            break;
            case "quit":
            wantToQuit = true;
            break;
```

```
            default:
            System.out.println("Dont get angry, calm down and try it again! ");
        }

    }

    return wantToQuit;

}

/**
 * Print exit some help information.
 * Here we print some stupid, cryptic message and a list of the
 * command words.
 */
private void printHelp()
{
    System.out.println("You need to get to the fourth floor ASAP!");
    System.out.println("Find the place with the nice view!");
    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 room, otherwise print an error message.
 * @param command
 */
private void goRoom(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.
    Room nextRoom=currentRoom.getExit(direction);
    if(nextRoom==null){
        System.out.println("There is no exit!");
    }else{

        currentRoom=nextRoom;

        System.out.println(currentRoom.getLongDescription());
```

```
            System.out.println("Items: " + currentRoom.getNameOfItem());

        }
    }

    private void goBack(){
     currentRoom = lastRoom;
     System.out.println(currentRoom.getLongDescription());
    }

    /**
     * "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
        }
    }

}
```

Room Class

```
import java.util.*;
/**
 * Class Room - a room in an adventure game.
 *
 * This class is part of the "World of Zuul" application.
 * "World of Zuul" is a very simple, text based adventure game.
 *
 * A "Room" represents one location in the scenery of the game.  It is
 * connected to other rooms via exits.  The exits are labelled 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.
 *
 * @author  Michael Kolling and David J. Barnes
 *          Modified by Hikari Sophia Stölzle and Nsimba Antonio
 *          Modified by Pavel Tsvyatkov and Stepan Burlachenko
 * @version 2019.07.05
 */
```

```
public class Room
{
    public String description;
    public HashMap<String, Room> exits;
    public ArrayList<Item> roomItems;


    /**
     * Create a room 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 Room(String description)
    {
        this.description = description;
        exits = new HashMap<>();
        roomItems = new ArrayList<Item>();

    }

    /**
     * Gets the items of the list
     * @ return itemNames String containing of all items found in the room
     */
    public String getItemsName(){
        String itemNames = "";
        if(roomItems.size() == 0){
            return " ";
        }
        for(Item items : roomItems){
            itemNames = " " + items.getNameOfItem();
        }
        return itemNames;
    }

    /**
     * Adds an item into the roomItems ArrayList.
     */
    public void addItem(Item item){
        roomItems.add(item);
    }

    /**
     * Define the exits of this room.  Every direction either leads
     * to another room or is null (no exit there).
     * @param north The north exit.
     * @param east The east east.
     * @param neighbor The south exit.
```

```
     * @param direction The west exit.
     */
    public void setExits(String direction, Room neighbor)
    {
        exits.put(direction, neighbor);
    }

    /**
     *
     */
    public Room getExit(String direction)
    {
        return exits.get(direction);
    }

    /**
     *
     */
    public String getExitString()
    {
        String returnString = "Exits:";
        Set<String> keys= exits.keySet();
        for(String key :keys)
        {
            returnString += " " + key;
        }
        return returnString;
    }

    /**
     * @return The description of the room.
     */
    public String getDescription()
    {
        return description;
    }

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

Command Class

```
/**
```

```
 * This class is part of the "World of Zuul" application.
 * "World of Zuul" 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  Michael Kolling and David J. Barnes
 *           Modified by Hikari Sophia Stölzle and Nsimba Antonio
 *           Modified by Pavel Tsvyatkov and Stepan Burlachenko
 * @version 2008.03.30
 */

public class Command
{
    private String commandWord; //CommandWord 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 commandword,
    {
        this.commandWord = firstWord; //commandWord;
        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() //CommandWord
    {
        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); //CommandWord.UNKNOWN);
    }

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

Parser Class

```
import java.util.Scanner;
import java.util.StringTokenizer;

/**
 * This class is part of the "World of Zuul" application.
 * "World of Zuul" 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  Michael Kolling and David J. Barnes
 *          Modified by Hikari Sophia Stölzle and Nsimba Antonio
 *          Modified by Pavel Tsvyatkov and Stepan Burlachenko
 * @version 2008.03.30
```

```
 */
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);
        }
        //return new Command(commands.getCommandWord(word1),word2);
    }
```

```
    /**
     * Print out a list of valid command words.
     */
    public void showCommands(){
        commands.showAll();
    }
}
```

CommandWords Class

```
import java.util.HashMap;

/**
 * This class is part of the "World of Zuul" application.
 * "World of Zuul" is a very simple, text based adventure game.
 *
 * This class holds an enumeration of all command words known to the game.
 * It is used to recognise commands as they are typed in.
 *
 * @author  Michael Kolling and David J. Barnes
 *          Modified by Hikari Sophia Stölzle and Nsimba Antonio
 *          Modified by Pavel Tsvyatkov and Stepan Burlachenko
 * @version 2008.03.30
 */

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


    /**
     * Constructor - initialise the command words.
     *
     */
    public CommandWords()
    {
       //nothing to do at the momenent
    }


    /**
     * Check whether a given String is a valid command word.
     * @return true if a given string is a valid command,
     *
```

```
     */
    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;

    }

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


}
```

Item Class

```
import java.util.HashMap;

/**
 * Class Item - an item in an adventure game.
 *
 * This class is part of the "World of Zuul" application.
 * "World of Zuul" is a very simple, text based adventure game.
 *
 * A "Item" represents an item in the scenery of the game.  It is
 * connected to the rooms.  The exits are labelled 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.
 *
 * @author Pavel Tsvyatkov and Stepan Burlachenko
 * @version 2019.07.05
 */
public class Item
{
    public String nameOfItem;
```

```
    public String descriptionOfItem;
    public int weightOfItem;

    /**
     * Creates an item  with name,description and weight.
     * @param nameOfItem The name of the item
     * @param descriptionOfItem The item's description.
     * @param weightOfItem The weight of the item
     */
    public Item(String nameOfItem, String descriptionOfItem, int weightOfItem)
    {
        this.nameOfItem = nameOfItem;
        this.descriptionOfItem = descriptionOfItem;
        this.weightOfItem = weightOfItem;

    }

     /**
     * @return The description of the item.
     */
    public String getNameOfItem()
    {
        return nameOfItem;
    }

    /**
     * @return The description of the item.
     */
    public String getDescriptionOfItem()
    {
        return descriptionOfItem;
    }

    /**
     * @return The weight of the item.
     */
    public int getWeightOfItem()
    {
        return weightOfItem;
    }
}
```