Pavel Tsvyatkov • Stepan Burlachenko
07 | 09 | 2019 • WH C 579

.

12 | Pick up and Carry

Index

Assignment

Task 1

Switch your project back with the group you swapped with last week. If you were working on the zuul-refactored game, find someone else who was doing that and swap with them, either pairwise or in a three-way swap. Read through the changes made and note down in your report what was changed. If you got your own project back, are you happy with the changes? If you got a zuul-refactored project, what was changed here from the refactored one? Are you missing anything?

All the changes done were pretty much the same changes we did ourselves as a part of the previous lab. There are some discrepancies in the names of methods but overall the ideas implemented are the same.

There is a new class called Item. The Location class now has two new methods: public String getItemString() and public void placeItem(Item item). There is also an ArrayList in the field and in the constructor called itemPlace. In the CommandWords class we now have a 'back' command in the validCommands string array. In the Game class we have a new field previousLocation as well as the following items wood, bottle, matches, bananas, water, disk, ropes, canvas, hammer, nails. The goBack method to be executed when the 'back' command is given.

```
/**
 * Brings the player back to the previous location
 */
private void goBack(){
    currentLocation = previousLocation;
    System.out.println(currentLocation.getLongDescription());
    System.out.println(currentLocation.getItemString());
}
```

There is also a test class LocationTest to test the methods of the Location class to add an item to a particular array list for a particular location.

```
/**
 * Sets up the test fixture.
 *
 * Called before every test case method.
 */
@Before
public void setUp()
{
    item1 = new Item("seashell","seashell", 2);
    location1 = new Location("seaShore");
    item2 = new Item("sand","fist full of sand", 1);
    location1.placeItem(item2);
    location1.placeItem(item1);
    location1.getItemString();
}
```

[30 min]

Task 2

If not already there, make sure there is an `Item` class in the project you get back. Items have names, descriptions, and weights. Make sure you can print out an item's description.

The Item class was already implemented. It was documented well so it was easy to understand the code. We were missing a field for an item's name so we added it and made sure to add a parameter for the name in the constructor's parameter list. Then we also had to go in the Game class and add the name of an item when it's being created.

In the Item class we had a method called "getDetailedDescription" that was returning the description and the weight of an item.

```
/**
 * Method for returning information about each item
 * @return a String that tells the player about the item
 * and it's weight
 */
public String getDetailedDescription(){
    String longDescription = description + ", weighs " + weight +"kg.";
    return longDescription;
}
```

We thought the idea was nice and went to see how it looks like when playing the game and we see the information about the items present in a location.

```
You are on the beach.
Exits: east south
You see: A few pieces of wood, weighs 3kg.
```

If there was more than 1 item, it was also working well.

```
> go east
You are near the shipwreck.
Exits: west
You see: An empty bottle, weighs 1kg. A box of matches, weighs 1kg.
```

[10 min]

Task 3

Implement a command "take" that has the name of the item as the second parameter. What happens if the item to be taken is not in the room?

Our initial idea to implement a command "take" was that we should look at the "goLocation" method and see how it works. We needed to print a message if there is no second word after "take" and if there is a second word we write something like “String item = command.getSeconWord()". However, when we did that we received an error from BlueJ that String cannot be converted to Item type, so we got stuck and started thinking how can we go about that problem. We knew that once we issue a "take" command, we need to check if the item is present in the room, if it’s present we need to remove it from the room and add it to our "inventory" that we called backpack and is an ArrayList. Everything we tried so far we were doing in the Game class and it didn’t seem right that it should be the one dealing with taking and dropping items. We decided to check the book and saw that it would be a good idea to create a "Player" class, which is going to take care of taking and dropping items later on. We went on to create a new Player class and implemented the fields first.

```
public class Player
{
    private Location currentLocation;
    private ArrayList<Item> backpack;
    private int backpackWeightLimit;
```

Our "player" had a starting location, a backpack which is an ArrayList to hold the items and the backpack has weight limit for later tasks that have to do with it.

```
public void takeItem(Command command)
{
    if(!command.hasSecondWord()) {
        // print a message if there is no second word
        System.out.println("What do you want to take?");
        return;
    }
    String item = command.getSecondWord();

    Item itemToPick = currentLocation.getItem(item);

}
```

By looking at the goLocation method, we created a similar takeItem method in the Player class. If there was no second word, we just simply print a message. If there is a second word, we now save the second word in a String and then create a new local variable "itemToPick" of type Item, which is now going to solve the issue we had earlier with saving a thing of type Item in a String.

The rest of the code we had to add is just making sure the item exists and adding it to our backpack so the complete "takeItem" method looked like this.

```
/**
 * This is the method that simulates taking an item from a room.
 * We take the second word, similarly to the goLocation method and
 * use that word as the item that we are trying to pick.
 * @param command the item we are trying to pick
 */
public void takeItem(Command command)
{
    if(!command.hasSecondWord()) {
        // print a message if there is no second word
        System.out.println("What do you want to take?");
        return;
    }
    String item = command.getSecondWord();

    Item itemToPick = currentLocation.getItem(item);

    if(itemToPick != null){
        backpack.add(itemToPick);
    }
    else{
        System.out.println("Item doesn't exist");
    }
}
```

If the item is not null, we add it to the backpack. If the player issued a non-existing item as second word then we just print a message "Item doesn't exist".

First, we made sure to have "take" as a valid command in our list of available commands.

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

Then we went to the Game class and added another else if for the take command. We now call the takeItem method and make sure to pass the "command" as a parameter, just the way we did with goLocation.

```
else if (commandWord.equals("take")){
    player.takeItem(command);
```

Then we went to play the game and try the command. At first we got a nullPointerException and we were wondering what's wrong.

```
java.lang.NullPointerException
        at Game.createLocations(Game.java:95)
        at Game.<init>(Game.java:22)
```

We checked what was the error and it was failing to create the locations.

We thought about it, but couldn't see our mistake. After checking the Game class over and over, we realized we forgot to create the player in the Game class constructor.

```
public Game()
{
    player = new Player();
    createLocations();
    parser = new Parser();
    backpack = new ArrayList<>();
}
```

We added "player = new Player();" to the constructor of the Game class and tried to play the game again. This time we didn't get a nullPointerException!

```
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
You see: A few pieces of wood, weighs 3kg.
> take wood
>
```

The command was working, but we noticed we forgot to print a message after taking an item, so it looks weird when we issue the command.

To fix that, we went in the Player class and added a new line of code to the takeItem method.

```
if(itemToPick != null){
    backpack.add(itemToPick);
    System.out.println("You picked up: " + itemToPick.getName());
```

Now when we take an item, we also print a message that shows the name of the item we picked up. We tested it in game and it worked well.

```
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
You see: A few pieces of wood, weighs 3kg.
> take wood
You picked up: wood
> go east
You are near the shipwreck.
Exits: west
You see: An empty bottle, weighs 1kg. A box of matches, weighs 1kg.
> take bottle
You picked up: bottle
```

[80 min]

Task 4

Implement the command "drop" to get rid of an item. "drop all" should do just that.

The idea behind the drop command is pretty similar to the ‘take’ one. The only difference is that it iterates over the contents of the backpack now (the array list). However, this time we had to foresee four cases: the item to drop is in the backpack, the item to drop is not in the backpack, there is something in the backpack and there is nothing in the backpack. For that, we used a bunch of conditional statements as well as a for-each and a for loop.

When we just drop all the items, it adds all the items to the arraylist of the items in the current location and then empties the array list of the backpack for the player.

```
String itemToDrop = command.getSecondWord();

if(itemToDrop.equals("all")){
    if(backpack.isEmpty()) {
        System.out.println("Your backpack is already empty.");
    }else {
        for(Item item : backpack){
            currentLocation.placeItem(item);
        }
        backpack.clear();
        System.out.println("You emptied the backpack.");
    }
}
```

When only one item is getting dropped, only this item is added to the array list of the items the current location has and then it is deleted from the backpack array list.

```
    for(int i = 0; i < backpack.size(); i++){
        if(backpack.get(i).getName().equals(itemToDrop)){
            currentLocation.placeItem(backpack.get(i));
            backpack.remove(i);
            System.out.println("You dropped " + itemToDrop);
        }
        else {
        System.out.println("You do not have " + itemToDrop +
        " in your backpack");
    }
```

Then, we needed to add the corresponding line to the prcocessCommand method of the Game class.

```
else if (commandWord.equals("drop")){
    player.dropItem(command);
}
```

Testing the possible setups resulted in success.

```
> drop
Drop what?
> drop bottle
You do not have bottle in your backpack
> drop wood
You dropped wood
> drop all
Your backpack is already empty.
>
```

Type input and press Enter to send to program

Then, we needed to check if the item dropped is added to the array list of items the current location has. This one worked too.

```
You are on the beach.
Exits: east south
You see: A few pieces of wood, weighs 3kg.
> take wood
You picked up: wood
> go south
You are deep into the jungles.
Exits: east south north west
You see: Some nice yellow bananas, weighs 2kg.
> drop wood
You dropped: wood
> look
You see: Some nice yellow bananas, weighs 2kg. A few pieces of wood, weighs 3kg.
```

[20 min]

Task 5

Implement the field to store the items currently carried by the player. How is the maximum weight determined? Does "take" know how to deal with this? You will probably need a method to ask if the current item can be taken by the player.

```
public class Player
{
    public Location currentLocation;
    private ArrayList<Item> backpack;
    private int backpackWeightLimit;
```

Our Player class has a field for the backpack weight limit.

In the constructor we also say that the backpack has a weight limit of 8.

```
/**
 * Constructor for creating a new player that
 * starts at currentLocation and has a backpack
 * with weight limit of 8kg.
 */
public Player()
{
    currentLocation = null;
    backpackWeightLimit = 8;
    backpack = new ArrayList<>();
}
```

Due to lack of time, we couldn't implement the rest of the changes to check if the item we are trying to take can be taken without exceeding the weight limit. Our main idea was that when trying to pick an item we should have an if statement that checks "if current weight of backpack + weight of the item we are trying to take is bigger than the backpackWeightLimit" and in that case we print a message saying we exceeded the limit. If we still had space for the item in our backpack, then we just take it. For this to be possible we would need an accessor method that gets the current weight limit of the backpack.

Evaluation

Pavel

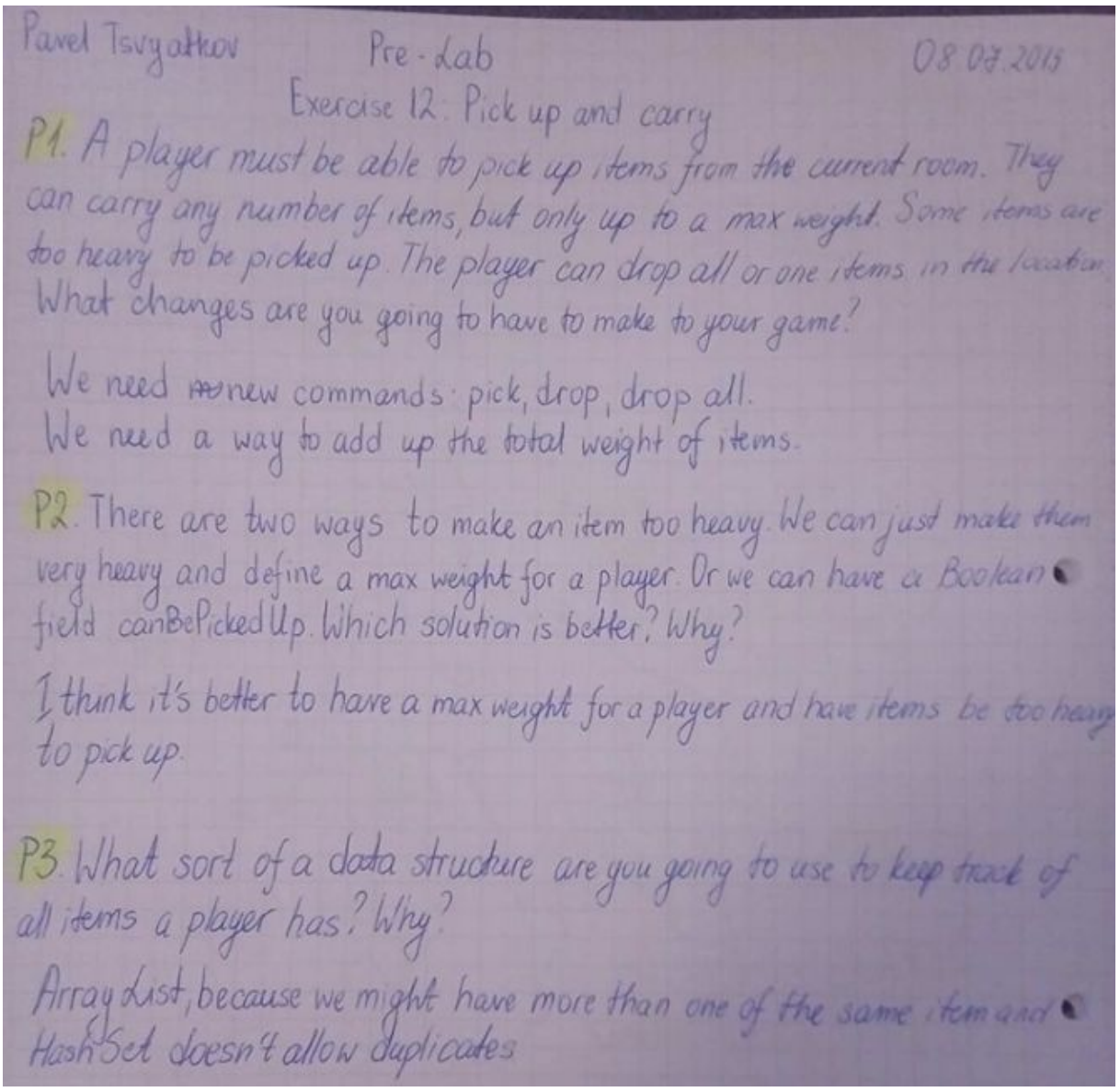
This lab exercise was very interesting, because we had to figure out how to deal with taking and dropping items. We did that by implementing a new class "Player". I learned more about how classes interact with each other and how tiny details can break our code or make us wonder why something is not working. It was important to do things step by step and test that the game is still working after we make changes in our code, so we know we aren't headed in the wrong direction. We had a few moments where we encountered the nullPointerException, but we figured out what we were doing wrong or forgot to implement.

Stepan

Here, we had to spend quite some time figuring the right way to implement the take/drop methods. It took awhile to understand that we probably needed a brand new class for that, this one was out main issue with this lab: finding the best possible way to solve the task. What this lab taught me is how classes interact with each other and how they call each other's methods inside their own methods.

Pre-Lab

Pavel



Pavel Tsvyatkov Pre-Lab 08.07.2019

Exercise 12: Pick up and carry

P1. A player must be able to pick up items from the current room. They can carry any number of items, but only up to a max weight. Some items are too heavy to be picked up. The player can drop all or one items in the location. What changes are you going to have to make to your game?

We need ~~no~~ new commands: pick, drop, drop all.
We need a way to add up the total weight of items.

P2. There are two ways to make an item too heavy. We can just make them very heavy and define a max weight for a player. Or we can have a Boolean field canBePickedUp. Which solution is better, Why?

I think it's better to have a max weight for a player and have items be too heavy to pick up.

P3. What sort of a data structure are you going to use to keep track of all items a player has? Why?

ArrayList, because we might have more than one of the same item and HashSet doesn't allow duplicates

Stepan

P1. Here are some informal specifications: A player must be able to pick up items from the current room. They can carry any number of items, but only up to a maximum weight limit. Some items are too heavy to be picked up. The player can drop one or all items in the current location. What changes are you going to have to make to your game? Don't program them yet, just specify the changes that need to be made to each class in writing.

New commands have to be implemented: pick, drop and drop all.
Also, somehow, we have to calculate the total weight of all the items the player currently has and give a message if the player tries to go over the limit. All this is most likely will be in class Game.

P2. There are (at least) two ways to make an item too heavy to pick up. We can just make them very heavy, and define what the maximum weight a player can pick up is. Or we can have a Boolean field canBePickedUp. Which solution do you think is better? Why?

Depends on the fact if these items are not 'pickable' by nature or they are 'pickable' by the player has to have no other items with them at that moment. For the former, a boolean should work, for the latter, it makes more sense to assign the max weight possible.

P3. What sort of a data structure are you going to use to keep track of all the items a player has? Why?

Probably hashset works best for this purpose because we do not care about the order of the items and all the objects we would have there are unique and never duplicates.

Appendix

Final Code

Game class

```
import java.util.ArrayList;

/**
 * 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 previousLocation;
        private ArrayList<Item> backpack;
        private Player player;
```

```
/**
* Create the game and initialise its internal map.
*/
public Game()
{
player = new Player();
createLocations();
parser = new Parser();
backpack = new ArrayList<>();
}

/**
* 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);
```

```
Item wood, bottle, matches, bananas, water, disk, ropes, canvas, hammer, nails;
//create the items
wood = new Item("wood","A few pieces of wood", 3);
bottle = new Item("bottle","An empty bottle", 1);
matches = new Item("matches","A box of matches", 1);
bananas = new Item("bananas","Some nice yellow bananas", 2);
water = new Item("water","Fresh water", 1);
disk = new Item("disk","Metal disks", 2);
ropes = new Item("ropes","Sturdy ropes", 2);
canvas = new Item("canvas","A large piece of canvas", 1);
hammer = new Item("hammer","A hammer", 2);
nails = new Item("nails","Some slightly rusty nails", 1);
//place items
beach.placeItem(wood);

shipwreck.placeItem(bottle);
shipwreck.placeItem(matches);

jungles.placeItem(bananas);

waterfall.placeItem(water);

volcano.placeItem(disk);

temple.placeItem(ropes);
temple.placeItem(canvas);

bunker.placeItem(hammer);
bunker.placeItem(nails);

player.setLocation(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(player.getLocation().getLongDescription());
System.out.println(player.getLocation().getItemString());
}

/**
* 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("take")){
player.takeItem(command);
}
else if (commandWord.equals("drop")){
player.dropItem(command);
}
```

```
else if (commandWord.equals("eat")){
eat();
}
else if (commandWord.equals("back")){
goBack();
}
// 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 = player.getLocation().getExit(direction);

if (nextLocation == null) {
System.out.println("There is no way!");
}
else {
previousLocation = player.getLocation();
player.setLocation(nextLocation);
```

```
        System.out.println(player.getLocation().getLongDescription());
        System.out.println(player.getLocation().getItemString());
        }
        }

        /**
        * Listing all the items that can be picked in the current location.
        */
        private void listItems()
        {
        System.out.println(player.getLocation().getItemString());
        }

        /**
        * 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.");
        }

        /**
        * Brings the player back to the previous location
        */
        private void goBack(){
        player.setLocation(previousLocation);
        System.out.println(player.getLocation().getLongDescription());
        System.out.println(player.getLocation().getItemString());
        }

        /**
        * "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;
import java.util.ArrayList;
/**
 * 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.

        public ArrayList<Item> itemPlace; //stores the items for 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<>();
        itemPlace = new ArrayList<>();
        }

        /**
        * 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);
}

/**
* Place Items in Locations
*/
public void placeItem(Item item){
itemPlace.add(item);
}

/**
* Prints a description of every item when the player enters a location
*/
public String getItemString(){
String itemString = "You see:";
for(Item item : itemPlace){
itemString += " " + item.getDetailedDescription();
}
```

```
        return itemString;
        }

        /**
        * This is the method that makes sure that after we have found an item
        * it returns the item as of type Item and not of type String
        * @param itemName the name of the item we have found
        * @return item the item we are taking
        */
        public Item getItem(String itemName){
        for(int i = 0; i < itemPlace.size(); i++){
        Item item = itemPlace.get(i);
        String nameOfItem = item.getName();
        if(nameOfItem.equals(itemName)){
                itemPlace.remove(i);
                return item;
        }
        }
        return null;
        }
}
```

Command 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", "eat", "take", "back", "drop"
        };

        /**
        * 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();
        }
}
```

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", "eat", "take", "back", "drop"
        };

        /**
        * 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();
        }
}
```

Item Class

```
    import java.util.ArrayList;
/**
 * Creates Items that have a description and a weight
 *
 * @version 2019-7-3
 */
public class Item
{
        private String name;
        //iformation displayed for the player
        private String description;
        //how much the item weighs
        private int weight;


        /**
        * Constructor for objects of class Item
        * @param a String that is used as information for the player
        * @param an int that determines how light or heavy the item is going to be
        */
        public Item (String name, String description, int weight){
        this.name = name;
        this.description = description;
        this.weight = weight;

        }

        /**
        * Method for returning information about each item
        * @return a String that tells the player about the item
        * and it's weight
```

```
	*/
	public String getDetailedDescription(){
	String longDescription = description + ", weighs " + weight +"kg.";
	return longDescription;
	}

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

	public String getName(){
	return name;
	}

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

Player Class

```
import java.util.ArrayList;
/**
 * The Player class holds the information about the current player as well as the player's current location and the items they have with them.
 *
 * @author  Pavel Tsvyatkov & Stepan Burlachenko
 * @version 2019
 */
public class Player
{
	public Location currentLocation;
	private ArrayList<Item> backpack;
	private int backpackWeightLimit;

	/**
	* Constructor for creating a new player that
```

```
* starts at currentLocation and has a backpack
* with weight limit of 8kg.
*/
public Player()
{
currentLocation = null;
backpackWeightLimit = 8;
backpack = new ArrayList<>();
}

/**
* Sets the current location of the player
* @param place the location of the player
*/
public void setLocation(Location place){
currentLocation = place;
}

/**
* Accessor method to get the current location
*/
public Location getLocation(){
return currentLocation;
}

/**
* This is the method that simulates taking an item from a room.
* We take the second word, similarly to the goLocation method and
* use that word as the item that we are trying to pick.
* @param command the item we are trying to pick
*/
public void takeItem(Command command)
{
if(!command.hasSecondWord()) {
// print a message if there is no second word
System.out.println("What do you want to take?");
return;
}
String item = command.getSecondWord();

Item itemToPick = currentLocation.getItem(item);

if(itemToPick != null){
backpack.add(itemToPick);
System.out.println("You picked up: " + itemToPick.getName());
}
else{
System.out.println("Item doesn't exist");
}
```

```
        }


        /**
        * This is the method that simulates dropping an item from our backpack.
        * We take the second word, similarly to the takeItem method and
        * use that word as the item that we are trying to drop.
        * @param command the item we are trying to drop
        */
        public void dropItem(Command command){
        if(!command.hasSecondWord()) {
        System.out.println("What do you want to drop?");
        return;
        }

        String itemToDrop = command.getSecondWord();

        if(itemToDrop.equals("all")){
        if(backpack.isEmpty()) {
                System.out.println("You don't have anything in the backpack.");
        }else {
                for(Item item : backpack){
                currentLocation.placeItem(item);
                }
                backpack.clear();
                System.out.println("You dropped everything from your backpack.");
        }
        }

        for(int i = 0; i < backpack.size(); i++){
        if(backpack.get(i).getName().equals(itemToDrop)){
                currentLocation.placeItem(backpack.get(i));
                backpack.remove(i);
                System.out.println("You dropped: " + itemToDrop);
        }
        else {
        System.out.println("You do not have " + itemToDrop + " in your backpack!");
        }
        }
}
}
```