Shozab anwar Siddique K21054573

The name and a short description of your game.

The description should include at least a user level description (what does the game do?) and a brief implementation description (what are important implementation features?).

The name of the game is “Pyramid hunt” . An archaeologist is trying to explore a pyramid. While exploring the pyramid, the archaeologist finds a lot of valuable items that he must collect in order to complete the game. However, in this process he is met by distinct beings in the pyramid which only slow him down. One of which is a ghost that will try to prevent the archaeologist from taking the riches of the pyramid. The only way to be safe from the ghost is to capture it using a device found in the pyramid. There is a mummy that will restrict access to a room unless you do what it tells you to do (retrieve an item in the pyramid for it).

A bullet point list of each base task you completed and how you completed it.

The game has several locations/rooms.

To create new rooms I must initialise the rooms in the createObjects method in the player class.

The player can walk through the locations. (This is already implemented in the code you are given.)

There are items in some rooms. Every room can hold any number of items. Some items can be picked up by the player, others can’t.

To create items, I first created an item class that takes the item name, description and weight as parameters in it’s constructor. This class then has a method to print the details of the item and return its item weight. To prevents some items from being picked up, I must simply set the item weight to a very large number greater than the player’s maximum allowed carrying weight.

The player can carry some items with him. Every item has a weight. The player can carry items only up to a certain total weight.

I have created an inventory class to store the list of items that the user is carrying. To carry items, I created a pickup method that removes the item from the current room and adds it to the inventory class. To implement a weight system a total weight and a maximum weight field in the player class is used. Total weight that the player can carry is 20. The total weight is compared with the item weight and the maximum weight before picking up an item to check whether the inventory has enough space to store the item. If the item is too heavy to pick up, you will not be able to pick it up.

The player can win. There has to be some situation that is recognised as the end of the game where the player is informed that they have won.

The player wins when he has collected all the valuable items. To do this I have created a win game method which checks whether an the player has collected all the valuable items in the pyramid. If this is the case, the play method in the game class is terminated by returning true when the game is complete.

Implement a command “back” that takes you back to the last room you’ve been in.

To do this I have made use of stacks so that just before the player changes room’s, the current room is pushed into the stack. Then by creating a back method, every time the word back is input, the element at the top of the stack is removed using the built-in pop method, so that the previous

element (the room) is now at the top of the stack. Then by simply assigning the current room the value of the room at the top of the stack, the player goes back to its previous location.

Add at least four new commands (in addition to those that were present in the code you got from us).

List of valid command words:

"go", "quit", "help", "look" , "back", "take", ”drop”, "inventory", "inspect", "drink", "talk", "give"

To create additional command words, we must first add the string of the word into the valid commands field in the class commandWords. Then associate each command word with a respective method.

The look command allows the user to view which room they are in and check what items/characters are in the room.

The take command’s method (Pickup method) firstly checks whether what item the user wishes to pick up. If this item is non-existent if so it will just ask the user what item it wants. If the user asks for a valid item, the method with firstly check if the current room contains that method using a while loop. If so by using an if loop, we can check whether the item weight in summation with total weight is less than the maximum allowed weight. If this is the case, the item will be removed from the current room and will be added to the inventory, adding on the weight of the item to the total carried weight. If the item is already carried by the player, then it will just return a message to the user letting them know that they are carrying it already.

The drop command’s methos (dropItem method) checks whether the user is even able to drop the item by firstly checking if the input is a valid item name and secondly whether the user is even carrying the item (if the item is in the inventory). If the item is in the inventory, the item is removed from the inventory and then added to whatever room the player is in so that they are able to pick it up later if they wish.

The inventory command displays to the player a list of items the user is carrying, their total weight and the space available. To do this a method in the inventory class is created that uses a for-each loop to output the set of items the user is carrying.

The inspect command display’s to the player the item description of the item the player wishes to inspect. The inspect method can work on any item whether its carried in inventory or not.

The drink command is used to ‘drink’ an item. The method checks whether the item can be drank or not by firstly checking if the item name is null and secondly whether its possible to drink the item. The item can be drank if the inventory contains the item or the room contains the item. So that you can’t drink the item if you are in a different room to where the item is and are not carrying the item.

The talk command is used to speak with a character in the room. The character will talk according to what the player is doing. Firstly, the method checks whether the room the player is currently in contains the specified character. If so it will allow the user to interact with the character otherwise the player won’t be able to converse with anyone.

The give command is used to give an item to a character to complete quests or to capture a character using a specific item. The method firstly checks whether the player is giving an existing item to an existing character. The method then checks whether the player is giving the item to the

correct character. If so, the method will remove the item from the player inventory and not add it back into the room, so it is no longer possible to pick it up.

A bullet point list of each challenge task you completed and how you completed it. This should include the challenge tasks listed in this document, as well as ones you came up with yourself.

Add characters to your game. Characters are people or animals or monsters – anything that moves, really. Characters are also in rooms (like the player and the items). Unlike items, characters can move around by themselves.

To create a character that moves around, I have firstly created a new method in the room class that generates a random possible point of exit. The way I have done is that I have used a for-each loop that for each possible direction that a character can move in from the current room, the exit is added to a local array list of rooms that holds the different exits for the current room. Then by initialising a local random variable, I have used the built-in "nextInt" method which takes in an upper limit (array list size), to randomly return a random point of exit. Then by importing a timer and timer task, I have made it so that the character moves to a random neighbouring door after every 30 seconds. To do this, I had to use the built-in scheduleAtFixedRate(TimerTask task, long delay, long period) method and then using the run method from the timer task to get a random exit.

I have also created a character that does not move around however it asks the player to retrieve an item from the pyramid to allow access to a new room to the player. The way it works is that the talk method firstly checks whether the player inventory contains the specific item "coffin" if it doesn't then it will tell the player to retrieve it. If the inventory contains the item, the character will ask the player to give the item to it. If the player gives the item to the character, the character will add a new item to the player's inventory called "key" will allow the player to unlock the room and gain access to a new room.

Extend the parser to recognise three-word commands. You could, for example, have a command give bread dwarf to give some bread (which you are carrying) to the dwarf.

Firstly, extend the parser to consider the first three words by creating a new string local variable in the method get command and add the implementation of the third word into the tokenizer. Then in the command class I have created a third word field and created method's that return the third word and check whether its null. Then in the give item method, it checks whether the third word is the correct character name and if the item can be given to the character. The input is in the form of; give – item – character name.

Add a magic transporter room – every time you enter it you are transported to a random room in your game.

To do this I firstly made the constructor of the room class take in a boolean value which checks whether a room is a transporter room. Then by creating an array list that stores all the different rooms and importing a random function, I can generate a next integer (taking the number of rooms as the limit which generates a random room from the array list. Then in the go room method, check if the next room is a transporter room, if so then call the random room method to generate a room and then assign this to the current room of the player.

Others. You can invent additional challenge tasks yourself. Several other challenge tasks are suggested in the textbook for you to get an idea of the level of difficulty that is appropriate.

Have made use of stacks so that the player has the ability to go repeatedly type back and eventually go back to their starting position. By pushing the current room into the stack before assigning next room to it so that every time the player moves to a different room, the previous room is added to the stack. Then to go back just use the pop method to remove the element at the top of the stack.

Added inventory command to display the set of items that the player is carrying.

Added an item called water that increases the maximum allowed weight that the player can carry after drinking it.

Use and enumerated type definition called CommandWord .

Make the character move between random neighbouring rooms in set time intervals.

For each of the following code quality considerations, give and explain an example in your project where you considered it: coupling, cohesion, responsibility-driven design, maintainability.

Coupling – Attempted to reference different classes as less as possible such that some classes are no longer reference one another at all e.g initially my item class referenced both my room class and my player class. It only references the player class now.

Cohesion – creating classes such as a player class reduce cohesion as the class is more well-defined for one specific task or entity.

Responsibility-driven design – initially my player class was responsible for generating the string of items in the rooms which is a breach of the responsibility driven design. As the room class holds information of the room, the string of items should be created in the room class.

Maintainability – My entire inventory class was initially squished into the pick up and drop methods. This made it much more difficult to comprehend what exactly the method was doing. By creating a separate class for it the method's became much more maintainable and cohesive.

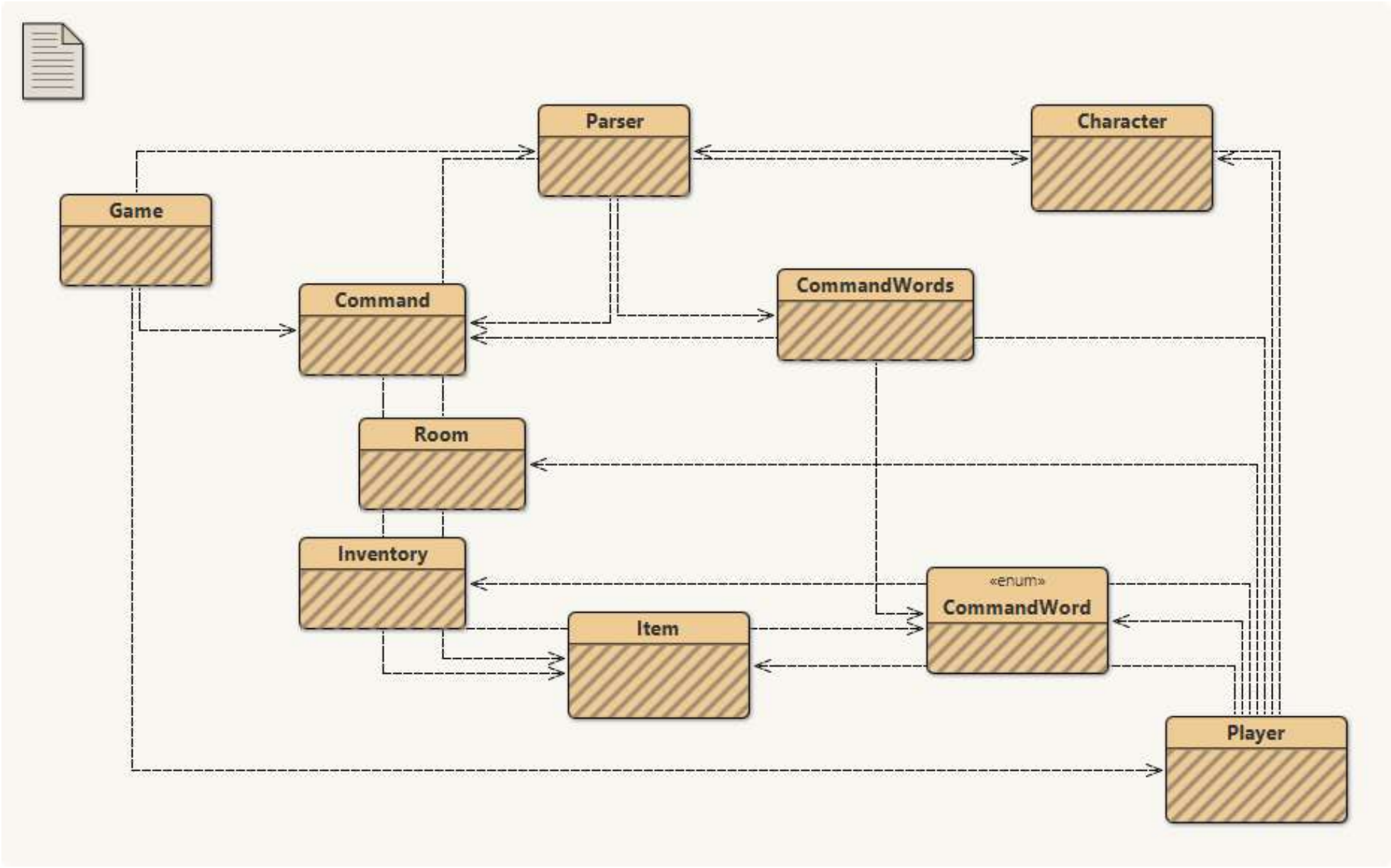
A walk-through of your game, consisting of the commands that need to be entered to complete/win the game.

Walk through does not take into consideration teleporter room and encounter with ghost as they are completely random cases.

Start at outside the pyramid, move west into hall of statues, go down into artefacts room, go south into the middle of nowhere, go west into the mummy's room, type "talk" to talk to mummy, go east to middle of nowhere, go south into artillery room, take bag, take trap, take tracker, go east into sanctuary, take water, go down into passageway, drink water, take coffin, go west into sewer, take bone, go north to treasure room, take treasure, back (x5), take torch, go west into mummy room, type talk, give coffin mummy, back, go north into artefacts room, take vase, go up, take statue.

Known bugs or problems (Note: for a bug in your code that you document yourself, you may not lose many marks — maybe none, if it is in a challenge task. For bugs that we find that you did not document you will probably lose marks.)

A copy of the source of all classes.


Parser
Character
Game
Command
CommandWords
Room
Inventory
«enum»
CommandWord
Item
Player

```java
import java.util.HashSet;
import java.util.Set;

/**
 * Write a description of class Item here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Item
{
    private String name;
    private String description;
    private int weight;
    /**
     * Creates an item that take in a name, description and a weight.
     * @param name The string name of the item
     * @param description The item's description.
     * @param weight The int weight value for an item.
     */
    public Item(String name, String description, int weight)
    {
        this.name = name;
        this.description = description;
        this.weight = weight;
    }

    /**
     * Print's the name, description and the weight for the item.
     */
    public void printItemDetails()
    {
        System.out.println("\nName: " + name);
        System.out.println("Description: " + description);
        System.out.println("Weight: " + weight);
    }

    /**
     * Return's the weight of the item
     * @return weight The int weight value for the item.
     */
    public int getWeight(){
        return weight;
    }

    /**
     * Retrieves the name of the item from the item object.
     * @param item The item object whose string name you are trying to get.
     * @return name The string name of the item.
     */
    public String getName(Item item){
        return name;
    }
}
```

```
import java.util.Stack;
import java.util.ArrayList;
import java.util.Set;
import java.util.HashMap;
import java.util.Random;
import java.util.Timer;
import java.util.TimerTask;

/**
 *  The player class is responsible for the interaction of the player with the the
rooms, items and characters.
 *
 * @author  Shozab Anwar Siddique
 * @version 2021.12.03
 */

public class Player
{
    private Parser parser;

    private Room currentRoom;
    private Stack<Room> stackRooms;

    private HashMap<String, Item>  stringToItem;

    private Inventory inventory;

    private Character mummy, ghost;

    private ArrayList<Room> rooms;

    private Room ghostRoom;
    /**
     * Create the game and initialise its internal map.
     */
    public Player()
    {
        createObjects();
        parser = new Parser();
        stackRooms = new Stack<>();
        inventory = new Inventory();
    }

    /**
     * Create all the rooms and link their exits together.
     */
    private void createObjects()
    {
        Room pyramidCentre, cave, mummyRoom, artillery, sanctuary, testRoom,
teleporter, artefact, hallOfStatues,outside,tomb,sewer,treasureRoom;

        Item boulder, statue, coffin, water, vase, bone, trap, bag, torch,
treasure, tracker, diamond;
```

```
        // create the rooms
        pyramidCentre = new Room("the middle of nowhere ", false);
        cave = new Room("a dark cave full of bats", false);
        mummyRoom = new Room("the mummy's resting room", false);
        artillery = new Room("a artillery room full of ancient weapons",false);
        sanctuary = new Room("a mysterious sanctuary", false);
        teleporter = new Room("a teleporter room", true);
        artefact = new Room("a room full of ancient artefacts",false);
        hallOfStatues = new Room("the hall of statues. Each item worth
millions",false);
        outside = new Room("the entrance to the pyramid", false);
        tomb = new Room("a narrow passegeway that holds a coffin resting on one of
the walls", false);
        sewer = new Room("a sewer system pathway",false);
        treasureRoom = new Room("a treasure room",false);

        rooms = new ArrayList<>();

        rooms.add(pyramidCentre);
        rooms.add(cave);
        rooms.add(artillery);
        rooms.add(sanctuary);
        rooms.add(artefact);
        rooms.add(hallOfStatues);
        rooms.add(outside);
        rooms.add(tomb);
        rooms.add(sewer);
        rooms.add(treasureRoom);

        // initialise room exits
        pyramidCentre.setExit("east", cave);
        pyramidCentre.setExit("south", artillery);
        pyramidCentre.setExit("west", mummyRoom);
        pyramidCentre.setExit("north", artefact);

        cave.setExit("west", pyramidCentre);

        mummyRoom.setExit("east", pyramidCentre);
        mummyRoom.setExit("north", teleporter);

        artillery.setExit("north", pyramidCentre);
        artillery.setExit("east", sanctuary);

        teleporter.setExit("north", mummyRoom);
        sanctuary.setExit("west", artillery);
        sanctuary.setExit("down", tomb);

        artefact.setExit("south", pyramidCentre);
        artefact.setExit("up", hallOfStatues);

        hallOfStatues.setExit("down", artefact);
        hallOfStatues.setExit("east", outside);

        outside.setExit("west", hallOfStatues);
```

```
        tomb.setExit("up", sanctuary);
        tomb.setExit("west", sewer);

        sewer.setExit("east", tomb);
        sewer.setExit("north",treasureRoom);

        treasureRoom.setExit("south", sewer);

        currentRoom = outside;  // start game outside

        boulder = new Item("Boulder", "Most likely used as a trap to prevent
people from \nexploring the pyramid. Looks inactive.", 100);
        water = new Item ("Water", "Water from this fountain apparently gives you
the \nability to carry more weight in your inventory",0);
        statue = new Item("Statue"," A statue of a pharaoh, this may be worth a
lot.",4);
        bag = new Item("Bag", "Used to carry items. I should take this.",0);
        coffin = new Item("Coffin", "Brightly painted coffin. Not what i expected
it to look like.", 21);
        trap = new Item("Trap", "Seems to be used to trap ghosts?? This is some
high tech equipment."
            +"\n (To trap a ghost you must give the trap to the ghost) ",2);
        torch = new Item("Torch", "Used to find my way through the dark",1);
        bone = new Item("Bone", "Remains of a skeleton in a worn out pirate
costume. Creepy.", 2);
        treasure = new Item("Treasure", "A treasure chest full of gold. Hit the
jackpot!",5);
        vase = new Item("Vase", "A broken vase. There might be someone in this
pyramid.",0);
        tracker = new Item("Tracker", "Highly advanced tech that can track
supernatural activity apparently", 5);
        diamond = new Item("diamond", "This diamond has to be one of the most
valuable items "
            +"\ni have seen in all my years as an archaeologist",0);

        mummy = new Character("Mummy", "An old wrapped up mummy. Maybe i can talk
with it.", false);
        ghost = new Character("Ghost", "A ghost that doesn't want anyone to get
inside his pyramid",false);
        ghostRoom = treasureRoom;

        cave.addItem("boulder",boulder);
        pyramidCentre.addItem("torch",torch);
        artillery.addItem("bag",bag);
        artillery.addItem("trap",trap);
        artillery.addItem("tracker",tracker);
        sanctuary.addItem("water",water);
        tomb.addItem("coffin",coffin);
        sewer.addItem("bone",bone);
        treasureRoom.addItem("treasure",treasure);
        artefact.addItem("vase",vase);
        hallOfStatues.addItem("statue",statue);

        mummyRoom.addCharacter(mummy);
```

```
        stringToItem = new HashMap<>();

        stringToItem.put("boulder",boulder);
        stringToItem.put("water", water);
        stringToItem.put("statue",statue);
        stringToItem.put("bag",bag);
        stringToItem.put("coffin",coffin);
        stringToItem.put("trap",trap);
        stringToItem.put("torch",torch);
        stringToItem.put("bone",bone);
        stringToItem.put("vase", vase);
        stringToItem.put("treasure",treasure);
        stringToItem.put("tracker", tracker);
        stringToItem.put("diamond",diamond);

        printWelcome();
    }

    /**
     * Print out the opening message for the player.
     */
    private void printWelcome()
    {
        System.out.println();
        System.out.println("Welcome to the World of Zuul!");
        System.out.println("The aim of the game is simple; explore the pyramid and
collect all the artifacts.");
        System.out.println("Type 'help' if you need help.");
        System.out.println("Pro Tip: Be sure to read the inspect the item before
picking it up");
        System.out.println();
        look();
        System.out.println();
    }

    /**
     * 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.
     */
    public boolean processCommand(Command command)
    {
        boolean wantToQuit = false;
        if(command.isUnknown()) {
            System.out.println("I don't know what you mean...");
            return false;
        }

        CommandWord commandWord = command.getCommandWord();
        switch (commandWord) {

            case UNKNOWN:
                System.out.println("I don't know what you mean...");
```

```
                break;

            case HELP:
                printHelp();
                break;

            case GO:
                goRoom(command);
                break;

            case QUIT:
                wantToQuit = quit(command);
                break;
            case LOOK:
                look();
                break;

            case BACK:
                goBack();
                break;

            case TAKE:
                pickUp(command);
                break;

            case DROP:
                drop(command);
                break;

            case INVENTORY:
                inventory.showInventory();
                break;

            case INSPECT:
                inspect(command);
                break;
            case DRINK:
                drink(command);
                break;

            case TALK:
                talk();
                break;

            case GIVE:
                giveItem(command);
                break;

        }
        // else command not recognised.
        return wantToQuit;
    }

    // implementations of user commands:
```

```
    /**
     * 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 are alone. You wander");
        System.out.println("around in the pyramid.");
        System.out.println();
        System.out.println("Your command words are:");
        parser.showCommands();
    }

    /**
     * Try to go in to one direction. If there is an exit, enter the new
     * room, otherwise print an error message.
     * @param command The user input used to determine where to go.
     */
    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 door!");
        }
        else if (nextRoom.isTransporterRoom()){
            stackRooms.push(currentRoom);
            currentRoom = getRandomRoom();
            System.out.println("What happened?? Have i been teleported?");
            look();
        }
        else {
            stackRooms.push(currentRoom);
            currentRoom = nextRoom;
            System.out.println(currentRoom.getLongDescription());
            currentRoom.getItemsInRoom();
            if(currentRoom.containsCharacter(mummy)){
                mummy.printCharacterDetails();
                System.out.println("Type talk....");
            }
        }
    }
```

```
    /**
     * Allows the player to give an item to a character.
     * @param commmand The user input use to determine what and who to give an
item to.
     */
    private void giveItem(Command command){
        if(!command.hasSecondWord()) {
            System.out.println("What would you like to give");
            return;
        }
        else if(!command.hasThirdWord()){
            System.out.println("Who would you like to give this item to?");
            return;
        }

        String characterName = command.getThirdWord();

        String giveItem = command.getSecondWord();

        Item item = getItemFromString(giveItem);

        while(giveItem != null)
        {
            if(item == null)
            {
                System.out.println("There is no item of that name!");
            }
            else if (giveItem.equals("coffin") && characterName.equals("mummy")){
                mummy.toggleSatisfied();
                inventory.remove(giveItem,item);
                System.out.println("You have given the item '"+giveItem+"'to the
"+characterName+".");
                talk();
            }
            else if (giveItem.equals("trap") && characterName.equals("ghost")){
                inventory.remove(giveItem,item);
                System.out.println("The "+characterName+" has been trapped inside
the "+giveItem+".");
                System.out.println("I will leave this ghost trapping device. It
could be dangerous to keep it");
            }
            else if (inventory.contains(giveItem))
            {
                System.out.println("I don't think the " + characterName + " would
want that.");
            }
            else {
                System.out.println("You are not carrying an item of that name!");
            }
            return;
        }
    }

    /**
     * Allows the user go back to it's previous room location.
```

```
     */
    private void goBack()
    {
        if(stackRooms.empty()){
            System.out.println("You can't go back further!");
        }
        else{
            currentRoom = stackRooms.pop();
            look();
        }
    }

    /**
     * Allows the user to pick up items in the room that they are in if the user
     * has a bag in their inventory.
     * @param command The user input used to check what item they want to take.
     */
    private void pickUp(Command command)
    {
        if(!command.hasSecondWord()){
            System.out.println("What would you like to take?");
        }

        String itemToPickUp = command.getSecondWord();

        Item item = getItemFromString(itemToPickUp);

        while(item != null && currentRoom.contains(item)){
            if(itemToPickUp.contains("tracker") && inventory.canPickUpItem(item)){
                inventory.addItem(itemToPickUp,item);
                currentRoom.remove(itemToPickUp,item);

                System.out.println(itemToPickUp + " has been added to your
inventory");
                System.out.println("\nThe device says that there is a ghost in
this pyramid.");
                System.out.println("It shows me where the ghost is at all
times!");

                setCharacterMovement();
            }
            else if(inventory.canPickUpItem(item)){
                inventory.addItem(itemToPickUp,item);
                currentRoom.remove(itemToPickUp, item);

                System.out.println(itemToPickUp + " has been added to your
inventory");
            }
            else if(!inventory.contains("bag")){
                System.out.println("I can't hold items until i find my bag!");
                return;

            }
            else{
                System.out.println("This item is too heavy for me to carry!");
```

```
            }
            return;
        }

        if(inventory.contains(itemToPickUp)){
            System.out.println("You are already carrying this item");
        }
        else{
            System.out.println("There is no such item of that name in this
room!");
        }
    }

    /**
     * Allows the user to drop desired items in the room that they are in.
     * @param The user input used to check what item they want to drop.
     */
    private void drop(Command command)
    {
        if(!command.hasSecondWord())
        {
            System.out.println("What would you like to drop?");
        }

        String itemToDrop = command.getSecondWord();

        Item item = getItemFromString(itemToDrop);

        while(itemToDrop != null)
        {
            if(item == null)
            {
                System.out.println("There is no item of that name!");
            }
            else if (inventory.contains(itemToDrop))
            {
                inventory.remove(itemToDrop,item);
                currentRoom.addItem(itemToDrop,item);
                System.out.println(itemToDrop + " has been dropped from your
inventory");
            }
            else
            {
                System.out.println("You are not carrying this item!");
            }
            return;
        }
    }

    /**
     * Allows the user to display the item descriptions of a specific item.
     * @param command The user input used to check what item they want to inspect.
     */
    private void inspect(Command command)
    {
```

```
        if(!command.hasSecondWord())
        {
            System.out.println("What would you like to inspect?");
        }

        String inspectItem = command.getSecondWord();

        Item item = getItemFromString(inspectItem);

        boolean canInspect = inventory.contains(inspectItem) ||
currentRoom.contains(item);

        if(item == null)
        {
            System.out.println("There is no item of that name!");
        }
        else if (canInspect == true){
            item.printItemDetails();
        }
    }

    /**
     * Allows the user to drink specific items only.
     * @param The user input used to check what item they want to drink.
     */
    private void drink(Command command)
    {
        if(!command.hasSecondWord())
        {
            System.out.println("What would you like to drink?");
        }

        String itemName = command.getSecondWord();

        Item item = getItemFromString(itemName);

        boolean canDrinkItem = inventory.contains(itemName) ||
currentRoom.contains(item);

        while(item != null && canDrinkItem)
        {
            if(itemName.equals("water"))
            {
                inventory.increaseCapacity();
                currentRoom.remove(itemName, item);
                inventory.remove(itemName,item);
                System.out.println("You can now carry a total of 35 weight!");
            }
            else
            {
                System.out.println("I can't drink that!");
            }
            return;
        }
```

```
        if(item == null)
        {
            System.out.println("There is no item of that name!");
        }
    }

    /**
     * "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
        }
    }

    /**
     * Display's what room the user is currently in and the items that
     * are present in that room.
     */
    private void look()
    {
        System.out.println(currentRoom.getLongDescription());
        currentRoom.getItemsInRoom();
    }

    /**
     * Return's the item object from its respective string.
     * @param item The string of the item.
     * @return The item object from the string.
     */
    private Item getItemFromString(String item)
    {
        return stringToItem.get(item);   // Used to prevent the use of additional
item fields.
    }

    /**
     * Simulates speech with the character mummy.
     */
    private void talk()
    {
        while(currentRoom.containsCharacter(mummy)){
            if(mummy.isSatisfied()){
                System.out.println("Thanks. Well i gave my word so i'll give you
the diamond."
                    +"\n\nYou received a new item!");
```

```
                inventory.addItem("diamond",getItemFromString("diamond"));
                getItemFromString("diamond").printItemDetails();
                return;
            }
            else if(!inventory.contains("coffin")){
                System.out.println("\nWhat are you here to do, take the riches of
the pyramid?. "
                    +"\nIt just happened to be that i am currently in possesion of
the most"
                    +"\nvaluable item in the pyramid, this massive diamond. You
won't get it from me."
                    +"\n\nHowever if you can find my coffin, i might consider
letting you in."
                    +"\nCoffin's tend to be heavy so you might to find the magic "
                    +"\nwater fountain of youth that gives you super strenght "
                    +"and makes you look younger."
                    +"\n\nGo north to get teleported. If you're lucky you'll get
closer to the coffin.");
                return;
            }
            System.out.println("I see you found my coffin. Give me the coffin if
you want the diamond.");
            return;
        }
        System.out.println("There is no one to talk to in this room!");
    }

    /**
     * End's the game after all the valuable items have been collected.
     * @return A boolean response of whether all the items have been collected.
     */
    public boolean winGame()
    {
        if((inventory.getSize() == (stringToItem.size()-4) &&
!inventory.contains("coffin") && !inventory.contains("tracker")) ){
            return true;
        }
        return false;
    }

    /**
     * Generates a random room from a collection of rooms.
     * @return The randomly selected room.
     */
    public Room getRandomRoom()
    {
        Random roomIndex = new Random();
        int numberOfRooms = rooms.size();
        return rooms.get(roomIndex.nextInt(numberOfRooms));
    }

    /**
     * Makes a particular character move in a randomly generated neighbouring room
     * after at set time intervals.
     */
```

```
    private void setCharacterMovement()
    {
        Timer timer = new Timer();
        if (inventory.contains("tracker")){
            timer.scheduleAtFixedRate(new TimerTask() {
                    @Override
                    public void run() {
                        ghostRoom = ghostRoom.getRandomExit();
                        System.out.println("The ghost has moved to " +
ghostRoom.getShortDescription());
                        if(currentRoom == ghostRoom){
                            System.out.println("A ghost has appeared in this
room!");
                            System.out.println("(To trap a ghost you must give the
trap to the ghost)");
                            System.out.println("\n   # Capture ghost if you have
the item 'trap'");
                            System.out.println("   # Change room's and come back
when you have the item 'trap' !! ");
                            timer.cancel();
                        }
                    }
                }, 0, 20000);
        }
    }
}
```

```
/**
 *  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 Kölling and David J. Barnes
 * @version 2021.12.03
 */

public class Game
{
    private Parser parser;
    private Player player;
    /**
     *
     */
    public static void main (String[] args){
        Game game = new Game();
        game.play();
    }

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

    /**
     *  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) {
            if(player.winGame() == true){
                finished = player.winGame();
                System.out.println("Congratulations, you have completed the
game!!");
```

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

```java
/**
 * Write a description of class Character here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Character
{
    private String name;
    private String characterDescription;
    private boolean isSatisfied;
    /**
     * Constructor for objects of class Item
     */
    public Character(String name, String characterDescription, boolean
isSatisfied)
    {
        this.name = name;
        this.characterDescription = characterDescription;
        this.isSatisfied = isSatisfied;
    }

    /**
     * An example of a method - replace this comment with your own
     *
     * @param  y  a sample parameter for a method
     * @return    the sum of x and y
     */
    public void printCharacterDetails()
    {
        System.out.println("There is a " + name + " in this room. "+
characterDescription);
    }

    /**
     *
     */
    public void toggleSatisfied()
    {
        isSatisfied = !isSatisfied;
    }

    /**
     *
     */
    public boolean isSatisfied(){
        return isSatisfied;
    }
}
```

```java
/**
 * 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 Kölling and David J. Barnes
 * @version 2016.02.29
 */

public class Command
{
    private String secondWord;
    private String thirdWord;
    private CommandWord commandWord;
    /**
     * Create a command object. First, second and third word must be supplied, but
     * either one (or all) 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.
     * @param thirdWord The third word of the command.
     */
    public Command(CommandWord commandWord, String secondWord, String thirdWord)
    {
        this.commandWord = commandWord;
        this.secondWord = secondWord;
        this.thirdWord = thirdWord;
    }

    /**
     * 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 CommandWord 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 == CommandWord.UNKNOWN);
    }

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

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

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

```java
/**
 * List of commands for the game
 *
 * @author Shozab Anwar Siddique K21054573
 * @version 2021.12.03
 */
public enum CommandWord
{
    GO("go"), QUIT("quit"), HELP("help"), UNKNOWN("?") ,LOOK("look"),
    BACK("back"), DROP("drop"),TAKE("take"),INVENTORY("inventory"),
    INSPECT("inspect"),DRINK("drink"),TALK("talk"),GIVE("give");

    private String commandString;

    /**
     * Initialise with the command string.
     * @param commandString The input command string.
     */
    CommandWord(String commandString)
    {
        this.commandString = commandString;
    }

    /**
     * Converts to string.
     * @return command word to string.
     */
    public String toString()
    {
        return commandString;
    }
}
```

```java
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 Kölling and David J. Barnes
 * @version 2016.02.29
 */

public class CommandWords
{
    private HashMap<String, CommandWord> validCommands;
    /**
     * Constructor - initialise the command words.
     */
    public CommandWords()
    {
        validCommands = new HashMap<>();
        for(CommandWord command : CommandWord.values()) {
            if(command != CommandWord.UNKNOWN) {
                validCommands.put(command.toString(), command);
            }
        }
    }

    public CommandWord getCommandWord(String commandWord)
    {
        CommandWord command = validCommands.get(commandWord);
        if(command != null){
            return command;
        }
        else{
            return CommandWord.UNKNOWN;
        }
    }

    /**
     * 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)
    {
        return validCommands.containsKey(aString);
    }

    /**
     * Print all valid commands to System.out.
     */
    public void showAll()
    {
```

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

```java
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;
import java.util.ArrayList;
import java.util.Random;

/**
 * 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.  For each existing exit, the room
 * stores a reference to the neighboring room.
 *
 * @author  Michael Kölling and David J. Barnes
 * @version 2016.02.29
 */

public class Room
{
    private String description;
    private HashMap<String, Room> exits; // stores exits of this room.
    private HashMap<String,Item> itemsInRoom;
    private HashSet<Character> charactersInRoom;
    private boolean isTransporter;
    /**
     * Create a room described "description". Initially, it has
     * no exits. "description" is something like "a kitchen" or
     * "an open court yard".
     * Takes into consideration whether the room is a teleporter
     * type or not.
     * @param description The room's description.
     * @param isTransporter The room type.
     */
    public Room(String description, boolean isTransporter)
    {
        this.description = description;
        this.isTransporter = isTransporter;
        exits = new HashMap<>();
        itemsInRoom = new HashMap<>();
        charactersInRoom = new HashSet<>();
    }

    /**
     * Define an exit from this room.
     * @param direction The direction of the exit.
     * @param neighbor  The room to which the exit leads.
     */
    public void setExit(String direction, Room neighbor)
    {
        exits.put(direction, neighbor);
    }
```

```
    /**
     * @return The short description of the room
     * (the one that was defined in the constructor).
     */
    public String getShortDescription()
    {
        return description;
    }

    /**
     * Return a description of the room in the form:
     *     You are in the kitchen.
     *     Exits: north west
     * @return A long description of this room
     */
    public String getLongDescription()
    {
        return "You are in " + description + ".\n" + getExitString();
    }

    /**
     * Return a string describing the room's exits, for example
     * "Exits: north west".
     * @return Details of the room's exits.
     */
    private String getExitString()
    {
        String returnString = "\nExits:";
        Set<String> keys = exits.keySet();
        for(String exit : keys) {
            returnString += " " + exit;
        }
        return returnString;
    }

    /**
     * Return the room that is reached if we go from this room in direction
     * "direction". If there is no room in that direction, return null.
     * @param direction The exit's direction.
     * @return The room in the given direction.
     */
    public Room getExit(String direction)
    {
        return exits.get(direction);
    }

    /**
     * Return a randomly generated room exit from all the possible exit points.
     * @return Random room exit.
     */
    public Room getRandomExit()
    {
        ArrayList<Room> possibleExits = new ArrayList<>();
```

```
        Random randomInt = new Random();

        Set<String> directions = exits.keySet();

        for(String direction : directions){
            possibleExits.add(getExit(direction));
        }

        int numberOfExits = possibleExits.size();
        return possibleExits.get(randomInt.nextInt(numberOfExits));
    }

    /**
     * Add's an item to the room.
     * @param itemString The string representation of the item.
     * @param item The actual item object.
     */
    public void addItem(String itemString,Item item)
    {
        itemsInRoom.put(itemString,item);
    }

    /**
     * Print's a list of items in the current room.
     */
    public void getItemsInRoom(){
        if(!itemsInRoom.isEmpty()){
            String returnString = "Items:";
            Set<String> itemsInRoomString = itemsInRoom.keySet();
            for(String item : itemsInRoomString) {
                returnString += " " + item;
            }
            System.out.println(returnString);
        }
        else{
            System.out.println("There are no items in this room.");
        }
    }

    /**
     * Checks whether a room contains a specified item.
     * @param item The item you are trying to check for.
     * @return A boolean answer to the check.
     */
    public boolean contains(Item item)
    {
        return itemsInRoom.containsValue(item);
    }

    /**
     * Removes the item object and the item string from the current room.
     * @param itemString The string representation of the item.
     * @param item The actual item object.
     */
```

```
    public void remove(String itemString, Item item)
    {
        itemsInRoom.remove(itemString, item);
    }

    /**
     * Add's a character to a specified room
     * @param character The character that is to be added to a room.
     */
    public void addCharacter(Character character)
    {
        charactersInRoom.add(character);
    }

    /**
     * Check's whether a specified character is in a room.
     * @param character The character that is being checked.
     * @return A boolean response to the check.
     *
     */
    public boolean containsCharacter(Character character)
    {
        return charactersInRoom.contains(character);
    }

    /**
     * Checks whether a room is of a teleporter type.
     * @return A boolean evaluation to the check.
     */
    public boolean isTransporterRoom()
    {
        return isTransporter;
    }
}
```

```java
import java.util.HashMap;
import java.util.Set;
/**
 * Write a description of class Inventory here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Inventory
{
    private HashMap<String,Item> items;
    private int maximumWeight;
    private int totalWeight;
    /**
     * Creates an inventory
     */
    public Inventory()
    {
        items = new HashMap<>();
        maximumWeight = 20;
        totalWeight = 0;
    }

    /**
     * Add's an item to the inventory class.
     * @param itemString A string representation of the item object.
     * @param item The actual item object that is to be added to the inventory.
     */
    public void addItem(String itemString, Item item)
    {
        items.put(itemString,item);
        totalWeight += item.getWeight();
    }

    /**
     * Remove's an item from the inventory class.
     * @param itemString The string representation of the item to be removed.
     * @param item The actual item object that is to be removed from the
inventory.
     */
    public void remove(String itemString,Item item)
    {
        items.remove(itemString,item);
        totalWeight -= item.getWeight();
    }

    /**
     * Return's a string of all the items in the inventory.
     * @return Details of the items in the inventory.
     */
    public String stringOfItems()
    {
        String returnString = "Inventory:";
        Set<String> itemsInRoom = items.keySet();
        for(String item : itemsInRoom) {
```

```
            returnString += " " + item;
        }
        return returnString;
    }

    /**
     * Return's the number of items that are present in the inventory.
     * @return The number of items.
     */
    public int getSize()
    {
        return items.size();
    }

    /**
     * Check's whether the inventory contains a specified item.
     * @param The string representation of the item we are checking for.
     * @return A boolean response to the check.
     */
    public boolean contains(String item)
    {
        boolean containsItem = false;
        if(items.containsKey(item))
        {
            containsItem = true;
        }
        return containsItem;
    }

    /**
     * Displays the items that are present in the inventory as well as
     * the total weight of the items and the remaining space.
     */
    public void showInventory()
    {
        if(items.isEmpty())
        {
            System.out.println("Currently not carrying anything!");
        }
        else
        {
            System.out.println(stringOfItems());
            System.out.println("Total Weight: "+totalWeight);

            int spaceLeft = maximumWeight - totalWeight;
            System.out.println("Remaining Space:" +spaceLeft);
        }
    }

    /**
     * Increases the maximum weight of items that can be carried by the inventory.
     */
    public void increaseCapacity()
    {
```

```
        maximumWeight = 35;
    }

    /**
     * Check's whether the inventory contains the item 'bag' if not
     * then it won't allow the user to carry items.
     * @param The item the user is trying to add to the inventory.
     * @return A boolean response to the check.
     */
    public boolean canPickUpItem(Item item)
    {
        boolean canPickUp = false;
        while(!items.containsKey("bag")){
            if(item.getName(item).equals("Bag")){
                canPickUp = true;
                System.out.println("Finally found my bag! Time to start the
hunt!");
            }
            return canPickUp;
        }

        if(totalWeight + item.getWeight() <= maximumWeight){
            canPickUp = true;
        }
        return canPickUp;
    }
}
```

```java
import java.util.Scanner;

/**
 * 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 Kölling and David J. Barnes
 * @version 2016.02.29
 */
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;
        String word3 = 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
                if(tokenizer.hasNext()){
                    word3 = tokenizer.next();    // get third word
```

```
                    // note: we just ignore the rest of the input line.
                }
            }
        }

        return new Command(commands.getCommandWord(word1), word2, word3);
    }

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

```
Project: zuul-better
Authors: Michael Kölling and David J. Barnes

This project is part of the material for the book

   Objects First with Java - A Practical Introduction using BlueJ
   Sixth edition
   David J. Barnes and Michael Kölling
   Pearson Education, 2016

This project is a simple framework for an adventure game. In this version,
it has a few rooms and the ability for a player to walk between these rooms.
That's all.

To start this application, create an instance of class "Game" and call its
"play" method.

This project was written as the starting point of a small Java project.

The goal is to extend the game:

  - add items to rooms (items may have weight)
  - add multiple players
  - add commands (pick, drop, examine, read, ...)
  - (anything you can think of, really...)

Read chapter 8 of the book to get a detailed description of the project.
```