
# BINUS UNIVERSITY
# BINUS INTERNATIONAL

## Assignment Cover Letter
(Individual Work)

| Student Information | |
|---|---|
| Full Name | Rachel Anastasia Wijaya |
| Student ID | 2502009646 |

| Course Information | |
|---|---|
| Course Name | Object Oriented Programming |
| Course Code | COMP6699001 |
| Class | L2CC |
| Major | Computer Science |
| Name of Lecturer | Jude Joseph Lamug Martinez |
| Type of Assignment | Final Project |

| Submission Pattern | |
|---|---|
| Due Date | 10 June 2022 |
| Submission Date | 9 June 2022 |

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

## Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating, and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity, and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

## Declaration of Originality

By signing this assignment, I understand, accept, and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student: Rachel Anastasia Wijaya

# Table of Contents

# Project Specification

The objective of this project is to create a parser game that uses the command line console to display and run the game.

## Game Overview

### a. Genre

Zwielicht is a parser game with a focus on exploration and character interaction. The current version is just a demo with a full map, but only 1 available NPC(Non-playable Character) to interact with.

### b. Game Concept

You play as an employee for an underground organization that specifically employs people in contracts with mythical beasts. The game is set in Voigt manor, the base of operations for the branch of the organization which deals with Norse/Germanic mythology. For the demo, it's mostly a free-roam, but as for the initial concept, it was going to be a puzzle game of sorts.

### c. Visual

As a parser game, Zwielicht has an emphasis on text, thus having little to no visuals.

### d. Game Flow Summary

When a new game is made, the player can freely explore the map and interact with any object they find. But if the player enters into a dialogue with an NPC, they will be forced to finish the interaction before regaining free movement. Since this is still a demo, there is no set game over or victory, so the player can freely explore and interact with the world after the dialogue interaction is over.

## Inspiration

### a. Norse Mythology

The main inspiration behind the game's story and concept was mythology from around the world, specifically about the mythical beasts and creatures found in them. Though I did initially plan to have more influences in the form of other regional mythos, such as Greek and Eastern mythology, the scope of it was unreasonable on the timeframe I was working with. So I ended up focusing on Norse mythology.

## b. Parser Games

While researching for game mechanics and dialogue systems, I came across a blog by Emily Short, which had lots of articles and resources discussing and explaining the ins-and-outs of dialogue systems, choices matter, and much more. In one article, she mentioned a parser game called *Color the Truth* by Brian Rushton, which became one of the main inspirations for the game.

## c. Visual Novels

Initially, I wanted to create a game where the combat system would be fighting, but with words and sentences. Instead of health points (HP), the player would need to keep up their social meter to win an encounter. But, as said previously, time constraints proved difficult to keep up with, and that idea was scrapped. I ended up settling for a more visual novel-like game, with dialogue choices and NPC interactions, but with the added exploration aspect too (however limited it is).

On top of that, I also wanted to give the game a "choices matter" feel. I found a paper by Ling Li and James Campbell on *Emotion Modeling and Interaction of NPCs in Virtual Simulation and Games* which made use of an attribute system with four primary feelings: anger, fear, relation, and happiness.

To add a bit more responsiveness to the game, each NPC (only one as of now) has special actions that can trigger if the corresponding attribute value is reached. For example, if the player reaches an NPC's anger trigger, they snap and end the conversation short. On the other hand, if an NPC's relation trigger is reached, they might gift the character with an item, so on and so forth.

# Solution Design

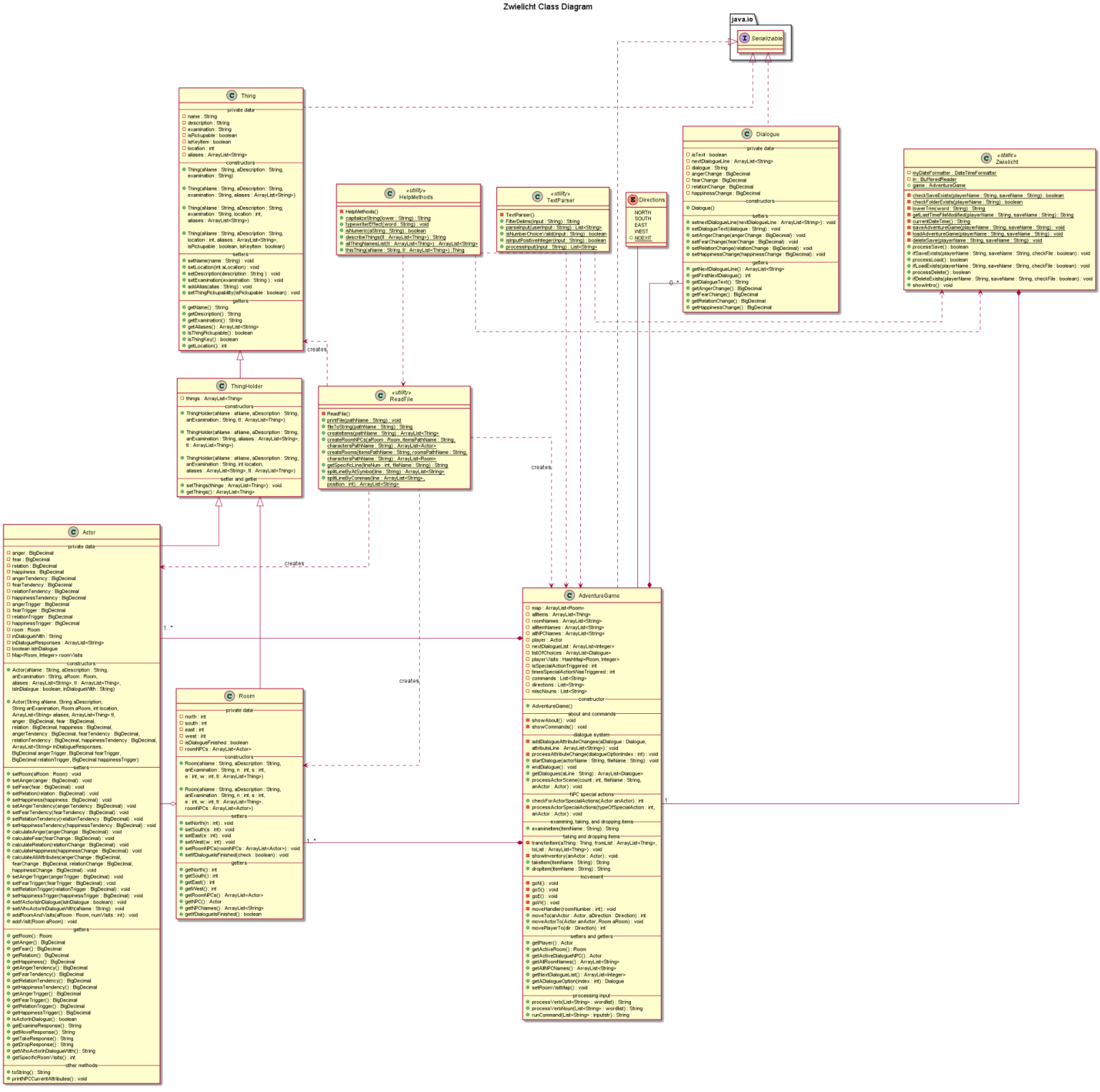
## Class Diagram

For a clearer picture, please refer to the .svg file in the GitHub Repository.

Explanation of the class diagram:

- **Thing Class**, the parent for creating items in the game. It is the parent class of the **ThingHolder Class** and implements the **java.io.Serializable interface.**
- **ThingHolder Class**, an extension of the **Thing Class**, used to store an ArrayList of Thing objects (used for objects that can hold things, like a person with an inventory of items).
- **Actor and Room Classes,** extensions of the ThingHolder Class, used to create the player character, NPCs, and rooms in the map.
- **Dialogue Class**, a class that is used to store dialogue lines and its attributes in the game.
- **HelpMethods, ReadFile, and TextParser Classes**, utility classes that are used to help read .txt files containing the information to create in-game objects (NPCs, dialogue, items, rooms).
- **Direction enum**, an enumeration for room exits in the game (north, south, east, west).
- **AdventureGame Class,** a class that holds the core of the game, handling and processing the user inputs and commands. It implements the java.io.Serializable interface.
- **Zwielicht Class**, a class that contains the main driver for the game and the save, load, and delete functionalities.

# Zwielicht Class Diagram

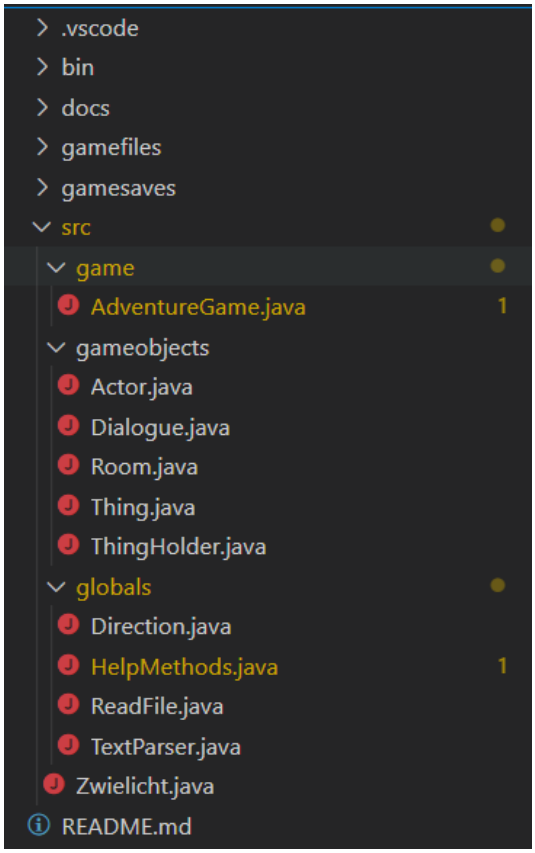


# Discussion

## Project Structure

The project is structured in this way:

- The docs folder is where documents related to the project, such as the map and project report.
- The gamefiles folder stores the text files that are used to create items, NPCs, and dialogue in-game
- The gamesaves folder stores all player save files
- The src folder is where game's source code is stored
    - The game folder stores the AdventureGame class
    - The gameobjects folder stores the Actor, Dialogue, Room, Thing, and ThingHolder classes
    - The globals folder stores the HelpMethods, ReadFile, TextParser classes and Direction enum
    - In the root of the src folder is ZwielichtGame, the game's main driver




## Game Mechanics

### a. Exploration

Exploration was arguably the simplest aspect when I was creating this game. The player can freely move from room to room, if they are not in a dialogue interaction. Each room in the map is filled with items and NPCs that the player can interact with.

- For items, the player can examine, take, or drop them. There are some things to consider:
    - If the item's isPickupable value is true, then it can be taken. If it is false, then it can't be taken.
    - If the item's isKeyItem value is true, then it can't be dropped (once it is taken or given to the player). If it is false, then it can be dropped.
    - If the item is in an NPC's inventory, it can't be examined.

    - The player can only examine NPCs and items that are assigned/currently in the active room.

- For NPCs, the player can only examine them, since a dialogue interaction is only triggered when a specific condition is met (in the demo, that condition is when the player first steps into the room named Edelmar's Office).
- Players can move using a move verb followed by a cardinal direction(north, south, east, west)

### b. Attribute System and NPCs

The attribute system was difficult to figure out initially, but I ended up using the figure from the aforementioned paper by Ling Li and James Campbell.


Fig. 1. Attributes of the four primary feelings (Color Plate 5)

It displays 4 spectrums of emotions that an NPC could be programmed to have. In addition, the paper also wrote about each NPC having their own emotional tendencies towards certain actions and events, like charm, intimidate, befriend, and impress. This was made to more accurately depict human emotional behavior, like how one person could be more prone to mood swings, while another could be calm and composed when faced with tough situations.

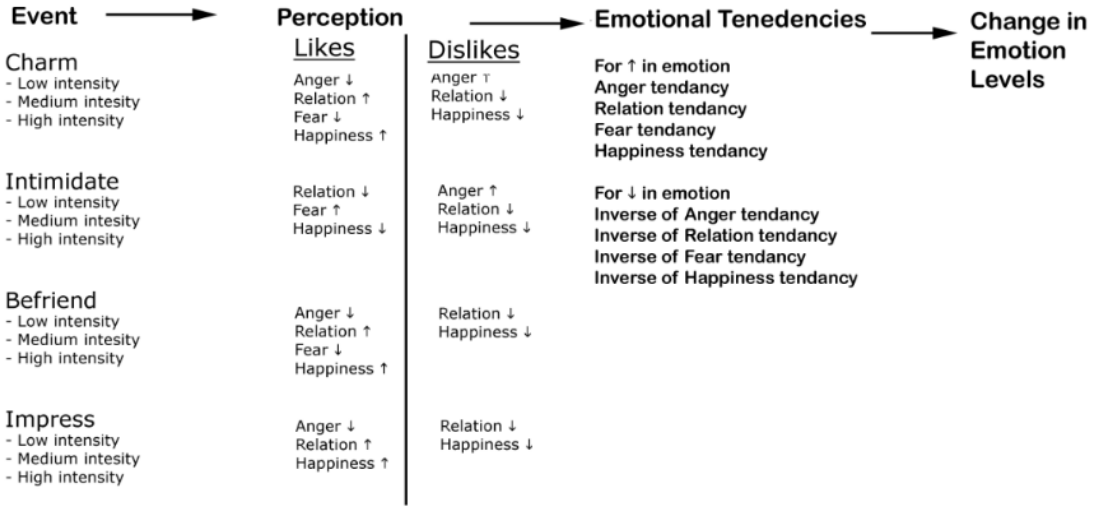



Fig. 3. Change in Emotion Levels

Since I knew the scope would be too overwhelming if I tried to fully implement the system, I went with a basic point system using the BigDecimal class. I also implemented the emotional tendencies, but in a much simpler way, by using multipliers.

For handling the NPC attribute system, I followed these main concepts:

- An NPC would have 4 attributes: anger, fear, relation, and happiness.
- An NPC would start with different base attributes.
- An NPC would have 4 emotional tendency multipliers for each of the attributes
- A positive (+) tendency value would mean the NPC is receptive towards actions that increase or decrease that tendency's attribute.
- A negative (-) tendency value would mean the NPC is resistant towards actions that increase or decrease that tendency's attribute.
- An NPC would have 4 attribute "triggers".
- An NPC would have special reactions if any of the attribute triggers are reached.

As an example, this is the NPC named Edelmar's attribute profile:

| **NPC Name** | Edelmar |
|---|---|
| **Base Anger** | 6 |

| | |
|---|---|
| **Base Fear** | 5 |
| **Base Relation** | 4 |
| **Base Happiness** | 4 |
| **Anger Tendency** | 0.25 |
| **Fear Tendency** | -0.5 |
| **Relation Tendency** | 0.75 |
| **Happiness Tendency** | -0.5 |
| **Anger Trigger** | 6.75 |
| **Fear Trigger** | 1.5 |
| **Relation Trigger** | 8 |
| **Happiness Trigger** | 8 |

From the profile, we can conclude that:

- Edelmar starts with relatively neutral attributes, ranging from 4-6
- Edelmar is receptive (+) towards the anger and relation attribute
- Edelmar is resistant (-) towards the fear and happiness attribute
- Edelmar's special action for anger will trigger when his anger reaches a value of 6.75 or higher
- Edelmar's special action for fear will trigger when his fear reaches a value of 1.5 or lower
- Edelmar's special action for relation and happiness will trigger when their respective attributes reach a value of 8 or higher

Attribute changes are calculated with the following formulas:

$If\ attribute\ change \neq 0$ :

- $If\ Actor\ attribute\ tendency > 0$

$$Resulting\ attribute\ value = Actor\ current\ attribute + (Actor\ attribute\ tendency * attribute\ change)$$

- $If\ Actor\ attribute\ tendency < 0$

$$Resulting\ attribute\ value = Actor\ current\ attribute - (Actor\ attribute\ tendency * attribute\ change)$$

As an example:

1. The player chooses an option that affects Edelmar's attributes like this:
    - Anger : + 3
    - Fear : - 1
    - Relation : 0
    - Happiness : - 2

2. The attributes will then be calculated as such:

    - Anger

    $$Since\ attribute\ change \neq 0\ and\ actor\ anger\ tendency\ > 0:$$
    $$Resulting\ anger\ = 6 + (0.25 * 3)$$
    $$Resulting\ anger\ = 6 + 0.75$$
    $$Resulting\ anger\ = 6.75$$

    - Fear

    $$Since\ attribute\ change \neq 0\ and\ actor\ anger\ tendency\ < 0:$$
    $$Resulting\ fear\ = 5 - ((-0.5) * (-1))$$
    $$Resulting\ fear\ = 5 - 0.5$$
    $$Resulting\ fear\ = 4.5$$

    - Relation

    $$Since\ attribute\ change\ = 0,\ no\ change\ in\ the\ relation\ attribute\ is\ made.$$
    $$Resulting\ relation\ = 4$$

    - Happiness

    $$Since\ attribute\ change \neq 0\ and\ actor\ happiness\ tendency\ < 0:$$
    $$Resulting\ happiness\ = 4 - ((-0.5) * (-2))$$
    $$Resulting\ happiness\ = 4 - 1$$
    $$Resulting\ happiness\ = 3$$

3. Edelmar's new attribute values are as such:
    - Anger : 6.75
    - Fear : 4.5
    - Relation : 4
    - Happiness : 3

### c. Dialogue

Though I planned on going for a similar dialogue system to *Color the Truth*, in which the player can glean more information by using topics they've found to initiate a conversation, I went against it, because of time constraints. I stuck with the traditional branching dialogue trees.

For handling the dialogue system, I followed these main concepts inspired from a simple example from the Unity Forums:

- A scene would be split into text and options.
- Texts are normal lines of dialogue and exposition.
- Options are where the player is presented with a maximum of 3 dialogue choices to choose.
- Options also have a chance of changing the involved NPC's attributes (for better, or for worse)

### d. Saves and Loads

Since one of the mechanics I wasn't able to implement in a previous game project of mine was the ability to save and load progress, I decided early on that this game would need one. Following a guide by Huw Collingbourne on how to make a text adventure game using the Java programming language, I used the java.io.Serializable interface to identify the Thing (its subclasses), Dialogue, and AdventureGame classes as serializable. This allows me to save its state, then restore (load) it at a later time. The save files use a .sav format and are saved in the gamesaves folder.

## Project Classes

### a. HelpMethods Class

**Included in:** globals package

The HelpMethods class is a static class that contains methods that are generally used in many of the other classes.

```
public final class HelpMethods{

    //private constructor to avoid instantiation
    private HelpMethods(){}
```

First, to avoid instantiation and overriding, I added a private constructor and made the class final. Then, I added the following methods:

- ❖ capitalizeString, a method to capitalize the first character of a String.

```
//public method to capitalize the first letter of a String
public static String capitalizeString(String lower){
    String output = lower.substring(beginIndex: 0, endIndex: 1).toUpperCase()  + lower.substring(beginIndex: 1);
    return output;
}
```

- isNumeric, a method to check if a String can be parsed into a numeric value.

```java
//check if a String can be parsed into an int
public static boolean isNumeric(String aString) {
    int intValue;

    if(aString == null || aString.equals(anObject: "")) {
        return false;
    }

    try {
        intValue = Integer.parseInt(aString);
        return true;
    } catch (NumberFormatException e) {
    }
    return false;
}
```

- describeThings, for returning a list of item names and each of their descriptions in a Thing ArrayList.

```java
//return the list + description of each item in the ThingList
public static String describeThings(ArrayList<Thing> tl){
    String s = "";
    if(tl.size() == 0){
        s = "There's nothing here, it's empty.";
    }else{
        for(Thing i: tl){
            if(i.isThingKey()){
                s += "KEY ITEM\n";
            }
            s += HelpMethods.capitalizeString(i.getName()) + ": \n" + i.getDescription() + '\n';
        }
    }

    return s;
}
```

- allThingNames, for returning a String ArrayList with all in-game items' names and aliases.

```java
//returns an ArrayList<String> of all item names in the game
public static ArrayList<String> allThingNamesList(ArrayList<Thing> tl){
    ArrayList<String> allThingNames = new ArrayList<>();
    ArrayList<String> thingAliases = new ArrayList<>();
    for(Thing t : tl){
        allThingNames.add(t.getName());
        thingAliases = t.getAliases();
        for(String alias : thingAliases){
            allThingNames.add(alias);
        }
    }
    return allThingNames;
}
```

- ❖ thisThing, for returning a specific Thing object from a Thing ArrayList when passed their name as a parameter.

```java
//return a specific thing from an ArrayList of Things
public static Thing thisThing(String aName, ArrayList<Thing> tl){
    Thing something = null;
    String thingName = "";
    ArrayList<String> thingAliases;
    for (Thing i : tl){
        thingAliases = i.getAliases();
        thingName = i.getName().trim().toLowerCase();
        if(thingName.equals(aName) || thingAliases.contains(aName)){
            something = i;
        }
    }
    return something;
}
```

b. ReadFile Class

**Included in:** globals package

The ReadFile class is a static class that contains methods that are used to read files to create the game's components, like the individual rooms, NPCs, items, and dialogue.

```java
public final class ReadFile {

    //private constructor to avoid instanciation
    private ReadFile(){}
```

Similar to the HelpMethods class, I added a private constructor and made the class final to avoid modification.

And below that are the methods that I added:

- printFile, a void method to print the entirety of a text file.

```
//method for printing out a txt file in its entirety
public static void printFile(String pathName){
    try{
        BufferedReader reader = new BufferedReader(new FileReader(pathName));
        String line = reader.readLine();

        //while it isn't the end of the file
        while(!line.equals(anObject: "ENDOFFILE")){
            System.out.println(line);
            line = reader.readLine();
        }

        reader.close();

    } catch (IOException e){
        System.out.println(x: "File could not be accessed, try again!");
    }

}
```

- fileToString, a method for returning a text file as a String

```
//method for returning a txt file as a String
public static String fileToString(String pathName){
    String msg = "";
    try{
        BufferedReader reader = new BufferedReader(new FileReader(pathName));
        String line = reader.readLine();
        while(!line.equals(anObject: "ENDOFFILE")){ //while it isn't the end of the file
            msg += line + "\n";
            line = reader.readLine();
        }
        reader.close();
        return msg;
    } catch (IOException e){
        System.out.println(x: "File could not be accessed, try again!");
    }
    return msg;
}
```

Before I explain the next few methods, I would like to explain how I organized the information in the text files. Also note that for every text file, you must add the phrase "**ENDOFFILE**" to signal the program that the text file has been fully read. For items, rooms, and NPC file formats, I followed and modified the format used by CRMinges in their game.

➔ For Items:
- Item name
- Starting room (where you would find the item initially)
- Item aliases (other names that can be used to name the item, separated by commas)
- If the item can be taken
- If the item is a key item
- Item description (this can extend for multiple lines of text)
- The phrase "**END**" (to signal to the program to read the **next item**'s information)

```
book on mythology
1
open book,mythology book,myth book,book about mythology,book about myths
true
false
A fairly thick book on the various mythos around the world. It's open and
looks recently read.
The page was opened on the section regarding Greek and Roman myths.
there are highlights on the page about Apollo.
END
```

➔ For NPCs:
- NPC name
- NPC starting room
- NPC aliases (separated by commas)
- NPC inventory items (separated by commas)
- NPC description (this can extend for multiple lines of text)
- The phrase "**ATTRIBUTES**" (to signal to the program to read NPC attributes)
- NPC base attributes (separated by commas)
- NPC emotional tendencies (separated by commas)
- NPC in dialogue responses (separated by @ symbols)
- NPC special action triggers (separated by commas)
- The phrase "**EXAMINE**" (to signal to the program to read NPC examination)
- NPC examination (this can extend for multiple lines of text)
- The phrase "**END**" (to signal to the program to read the **next NPC**'s information)

```
Edelmar
1
edel,el
pocketwatch,fountain pen,hunting dagger
Heir to the Voigt family. He's been part of this organization for around 8 years.
Quite a while, compared to you. Promoted to overseer of this branch 3 years ago.
With glares people say is as frigid as Fimbulvetr, he's known to be difficult to
approach, let alone make conversation with. (But that doesn't stop some)

Right now, he just looks tired.
ATTRIBUTES
6,5,4,4
0.25,-0.50,0.75,-0.50
"Honestly, you are in a conversation, please focus."@"Really now, you're trying to leave in the middle of a conversation?"@"Put that back, stop acting like a child."@"Pick that back up, quickly please."
6.75,1.5,8,8
EXAMINE
Upon closer inspection, the cracks in the glass show clearer.

His hands tremble with every stroke of ink on paper. Dark circles under his eyes
concealed hastily with makeup. Each breath seems shallow and weighty. Like someone pulling an
all-nighter to catch up with a morning deadline.
END
ENDOFFILE

```

➔ For Rooms

- Room name
- Room north exit (room index), south exit (room index), east exit (room index), west exit (room index). If a room index is equal to -1, it means no exit
- Room description (this can extend for multiple lines of text)
- Room examination (this can extend for multiple lines of text)
- The phrase "**END**" (to signal to the program to read the **next Room**'s information)

```
West Hallway
0
4,5,7,1
The west hallway of the estate. The tiles are practically seared into my brain
with how many times you've had to come to *his* office.
EXAMINE
To the north are the restrooms. To the south... someone's office. To the east
is back to the central hall, and west is.. well. Edelmar's office.
END
```

➔ For Dialogue Interactions

Since I wanted to implement branching dialogues, I needed to separate lines that are just text and lines that are actually dialogue options for the player. To accomplish that, I followed and modified a simple text format proposed by a user from the Unity Forums.

An example of the modified format:

```
text@"You know, when I said "feel free to visit me whenever", I didn't mean it literally."@2,28,31
option@"C'mon, you should know me better by now, El."@-1,0,1,-1@3
text@You hear him scoff. After a few more seconds of scribbling and a final flourish of@4
text@his signature, he sets down his pen and finally graces you with his full attention.@5
text@"So, what is it you're in need of this time?"@6,24
option@"Can't a friend just visit for the sake of it? I thought you'd be bored, cooped up all day in this stuffy office."@-1,-1,2,2@7
```

Separated by the @ symbol, a line in the text file consists of:

- For text (normal dialogue lines, not a dialogue choice)
  - text (identifies that it is a line of text)
  - Dialogue (the line of dialogue to show)
  - Line number (where this line of dialogue leads to next, if it leads to a choice, it will have more than one link)
- For options (a dialogue choice for the player)
  - option (identifies that this is a dialogue option for the player)
  - Dialogue (the dialogue option to show)
  - Attribute changes when this option is chosen (consists of numeric values representing anger, fear, relation, and happiness changes, in that order)
  - Line number (where this line of dialogue leads to next)

Now that the format has been explained, I will continue onto the rest of the ReadFile methods.

- createItems, a method for creating the in-game items by reading a file from the given path name and storing them in a Thing ArrayList. The method goes through the text file in a step-by-step process, the BufferedReader going through each line of text according to the format specified above. The method returns an error message if an exception is encountered.

```
//method to read and create the game items
public static ArrayList<Thing> createItems(String pathName){
    try{
        BufferedReader reader = new BufferedReader(new FileReader(pathName));
        String line = reader.readLine();
        ArrayList<Thing> items = new ArrayList<Thing>();

        while(!line.equals(anObject: "ENDOFFILE")){
            //item name
            String name = line;
            name = name.trim().toLowerCase();

            line = reader.readLine();

            //item location
            int location = Integer.parseInt(line);
            line = reader.readLine();

            //item aliases (other names for the item)
            ArrayList<String> aliases = new ArrayList<>(Arrays.asList(line.split(regex: ",")));
            line = reader.readLine();

            //is the item pickupable?
            boolean isPickupable = Boolean.parseBoolean(line);
            line = reader.readLine();
```

```
            //is the item a key item?
            boolean isKeyItem = Boolean.parseBoolean(line);
            line = reader.readLine();

            //item examination (detailed description when you examine a location)
            String description = "";
            while(!line.equals(anObject: "END")){
                description += line + '\n';
                line = reader.readLine();
            }

            //creating a new Thing object
            Thing anItem = new Thing(name, description, location, aliases, isPickupable, isKeyItem);
            //adding the item to the list
            items.add(anItem);

            line = reader.readLine();
        }

        reader.close();
        return items;

    } catch (IOException e){
        System.out.println(x: "File could not be accessed, try again!");
    }
    return null;
}
```

- createRoomNPCs, a method for creating the room's NPCs (and their inventory items) by reading a file from the given path name and returning them in an **Actor ArrayList**. It also takes a Thing ArrayList, containing all the available items in the game. This method also returns an error message if an exception was found when reading the file.

```
public static ArrayList<Actor> createRoomNPCs(Room aRoom, ArrayList<Thing> allItems, String charactersPathName){

    try{
        BufferedReader reader = new BufferedReader(new FileReader(charactersPathName));
        String line = reader.readLine();
        ArrayList<Actor> NPCs = new ArrayList<>();

        while(!line.equals(anObject: "ENDOFFILE")){
            //actor name
            String name = line;
            name = name.trim().toLowerCase();

            line = reader.readLine();

            //actor location
            int location = Integer.parseInt(line);
            line = reader.readLine();

            //actor aliases (other names for the actor)
            ArrayList<String> aliases = new ArrayList<>(Arrays.asList(line.split(regex: ",")));
            line = reader.readLine();

            //actor inventory
            ArrayList<String> actorInventoryList = new ArrayList<>(Arrays.asList(line.split(regex: ",")));
            ArrayList<Thing> actorInventory = new ArrayList<Thing>();
            for(String itemName : actorInventoryList){
                actorInventory.add(HelpMethods.thisThing(itemName, allItems));
            }
            line = reader.readLine();
```

createNPCRooms method continued:

```
//actor description
String description = "";
while(!line.equals(anObject: "ATTRIBUTES")){
    description += line + '\n';
    line = reader.readLine();
}
line = reader.readLine();

//actor attributes
ArrayList<String> attributesList = new ArrayList<>(Arrays.asList(line.split(regex: ",")));
//base anger
BigDecimal baseAnger = new BigDecimal(attributesList.get(index: 0));
//base fear
BigDecimal baseFear = new BigDecimal(attributesList.get(index: 1));
//base relation
BigDecimal baseRelation = new BigDecimal(attributesList.get(index: 2));
//base happiness
BigDecimal baseHappiness = new BigDecimal(attributesList.get(index: 3));

line = reader.readLine();

//actor tendencies
ArrayList<String> tendenciesList = new ArrayList<>(Arrays.asList(line.split(regex: ",")));
//angertendency (How prone they are to anger)
BigDecimal angerTendency = new BigDecimal(tendenciesList.get(index: 0));
//fearTendency (How prone they are to fear)
BigDecimal fearTendency = new BigDecimal(tendenciesList.get(index: 1));
//relationTendency (How prone they are to relation)
BigDecimal relationTendency = new BigDecimal(tendenciesList.get(index: 2));
//happinessTendency (How prone they are to happiness)
BigDecimal happinessTendency = new BigDecimal(tendenciesList.get(index: 3));
line = reader.readLine();

//examine response when in dialogue
ArrayList<String> dialogueResponseList = new ArrayList<>(Arrays.asList(line.split(regex: "@")));
line = reader.readLine();

//actor's special action attribute triggers
ArrayList<String> specialActionTriggers = new ArrayList<>(Arrays.asList(line.split(regex: ",")));
//base anger
BigDecimal angerTrigger = new BigDecimal(specialActionTriggers.get(index: 0));
//base fear
BigDecimal fearTrigger = new BigDecimal(specialActionTriggers.get(index: 1));
//base relation
BigDecimal relationTrigger = new BigDecimal(specialActionTriggers.get(index: 2));
//base happiness
BigDecimal happinessTrigger = new BigDecimal(specialActionTriggers.get(index: 3));

line = reader.readLine();
```

createNPCRooms method continued:

```
            //item examination (detailed description when you examine a location)
            String examination = "";
            while(!line.equals(anObject: "END")){
                examination += line + '\n';
                line = reader.readLine();
            }

            //creating a new Actor object
            Actor anActor = new Actor(name, description, examination, aRoom, location, aliases, actorInventory, baseAnger, baseFear, baseRelation, baseHappiness, angerTendency,
                                      fearTendency, relationTendency, happinessTendency, dialogueResponseList, angerTrigger, fearTrigger, relationTrigger, happinessTrigger);
            //adding the Actor object to the list
            NPCs.add(anActor);

            line = reader.readLine();
        }

        reader.close();
        return NPCs;

    } catch (IOException e){
        System.out.println(x: "File could not be accessed, try again!");
    }
    return null;
}
```

- createRooms, a method for creating the in-game rooms (and their items and NPCs) by reading a file from the given path name and returning them in a **Room ArrayList**. It calls both the createItems and createNPCRooms methods. This method also returns an error message if an exception was found when reading the file.

```
//method to read and create the game rooms
public static ArrayList<Room> createRooms(String itemsPathName, String roomsPathName, String charactersPathName){

    //calling the createItems to create all in-game items
    ArrayList<Thing> allItems = createItems(itemsPathName);

    try{
        BufferedReader reader = new BufferedReader(new FileReader(roomsPathName));
        String line = reader.readLine();
        ArrayList<Room> rooms = new ArrayList<>();

        while(!line.equals(anObject: "ENDOFFILE")){
            //location name
            String name = line.trim();
            line = reader.readLine();

            //location index
            String index = line.trim();
            int numIndex = Integer.parseInt(index);
            line = reader.readLine();

            //location exits
            ArrayList<String> neighbors = new ArrayList<>(Arrays.asList(line.split(regex: ",")));

            int n = Integer.parseInt(neighbors.get(index: 0));
            int s = Integer.parseInt(neighbors.get(index: 1));
            int e = Integer.parseInt(neighbors.get(index: 2));
            int w = Integer.parseInt(neighbors.get(index: 3));
            line = reader.readLine();
```

createRooms method continued:

```
            //location description
            String description = "";
            while(!line.equals(anObject: "EXAMINE")){
                description += line + '\n';
                line = reader.readLine();
            }
            line = reader.readLine();

            //location examination (detailed description when you examine a location)
            String examination = "";
            while(!line.equals(anObject: "END")){
                examination += line + '\n';
                line = reader.readLine();
            }

            //finding the corresponding items to be put in the room
            ArrayList<Thing> roomItems = new ArrayList<Thing>();
            for(Thing i : allItems){
                if(i.getLocation() == numIndex ){
                    roomItems.add(i);
                }
            }

            //creating a new Room object
            Room aRoom = new Room(name, description, examination, n, s, e, w, roomItems);
            //creating the room NPCs with createRoomNPCs
            ArrayList<Actor> roomNPCs = createRoomNPCs(aRoom, allItems, charactersPathName);
            //adding the room NPCs to the Room object
            aRoom.setRoomNPCs(roomNPCs);
            //adding the Room object to the list
            rooms.add(aRoom);

            line = reader.readLine();

        }

        reader.close();
        return rooms;

    } catch (IOException e){
        System.out.println(x: "File could not be accessed, try again!");
    }
    return null;

}
```

The methods below are used while processing a dialogue interaction, they're used to help parse the text file and return the needed values.

- ❖ getSpecificLine, a method for returning one specific line of text in a text file, as a **String.**

```
//getting a specific line in a txt file
public static String getSpecificLine(int lineNum, String fileName) throws IOException{
    //the actual line number needs to be subtracted by one to retrieve the correct line
    String line = Files.readAllLines(Paths.get(fileName)).get(lineNum - 1);
    return line;
}
```

- ❖ splitLineByAtSymbol, a method for splitting a line of text by using the at (@) symbol and returning it as a **String ArrayList**

```
//parsing a line in a txt by splitting it according to the regex (@)
public static ArrayList<String> splitLineByAtSymbol(String line){
    ArrayList<String> splitLine = new ArrayList<>(Arrays.asList(line.split(regex: "@")));
    splitLine.add(splitLine.get(splitLine.size()-1).trim());
    return splitLine;
}
```

- ❖ splitLineByCommas, a method for splitting a **String ArrayList**, specifically a dialogue line, by using the comma (,), returning it as a **String ArrayList.** It can also pass a negative position to find an index from the back of the ArrayList.

```
//splitting an ArrayList<String> by commas
public static ArrayList<String> splitLineByCommas(ArrayList<String> line, int position){
    ArrayList<String> splitLine = new ArrayList<>();
    //for finding indexes from the back of the ArrayList
    if(position < 0){
        List<String> result = Arrays.asList(line.get(line.size() + position).split(regex: ","));
        splitLine.addAll(result);
    }
    //for finding indexes normally (counting from the beginning)
    else{
        List<String> result = Arrays.asList(line.get(position).split(regex: ","));
        splitLine.addAll(result);

    }
    return splitLine;
}
```

c. TextParser Class

**Included in:** globals package

The TextParser class is a static class that contains methods that are used to parse user input into commands that the program can understand.

```
public final class TextParser {

    //private constructor to avoid instantiation
    private TextParser(){}
```

Similar to the HelpMethods and ReadFile classes, I added a private constructor and made the class final to avoid modification.

This was mostly based off of a text parser program for the game *Christmas Adventure*, made by Myre Mylar, but it also adds elements from an adventure game made by Huw Collingbourne. I added more commands to parse and it also has some additional methods for specific inputs (for example numeric choices for the dialogue choices).

The TextParser class has the following methods:

- filterDelims, a method that splits a string into tokens, filters out the set delimiters, rejoins the list of words and returns it as a **String**.

```
//method to filter out any punctuation/special characters
public static String FilterDelims(String input) {
    //setting the characters + white space as delimiters
    String delims = " \t,.:;?!\"'><()[]{}/|-_=+&*%$#@~`";
    List<String> wordList = new ArrayList<>();
    StringTokenizer tokenizer = new StringTokenizer(input, delims);
    String t;

    //adding the leftover words into the wordList
    while (tokenizer.hasMoreTokens()) {
        t = tokenizer.nextToken();
        wordList.add(t);
    }

    //rejoining the wordList back into a String
    String listString = String.join(delimiter: " ", wordList);
    return listString;
}
```

- parseInput, the main method used to parse the user input into commands.

Since it's quite a long method and all the commands are parsed in a similar way, I'll just present one example of the method parsing a command to examine something.

1. User inputs "inspect book of spells"
2. Input text is converted to lowercase.
3. Lowercase input text is split by whitespace into: ["inspect", "book", "of", "spells"]
4. The first index ("inspect") is found to be an examine action, so the value of foundExamineWords becomes true.
5. Since only 1 examine word was found, the remainingWordsindex value is 1

6. Since foundExamineWords is true, the remaining words in the list are processed and returned in the form of a command (“examine”) and an object (“book of spells”)

❖ Initial Section

In the first section of the method, the user input is converted into lowercase to keep the convention of inputs being not case sensitive.

Then, the initialization of boolean values that check whether a specific word related to a valid command is found.

Below that are String Lists that contain the possible inputs that are recognized by the program. I tried to include any common phrases and variations for each type of action.

The variable **remainingWordsindex** is used to check whether 1 or 2 command words were found. For example, “check” would count as 1 command word, while “look at” would count as 2 command words.

The variable **remainingWords** is used to store the String of the noun that was passed alongside the command. For example, “painting”, “fountain pen”, and so on.

The **result** variable is a String list that will contain the final list of commands for the game program. For example, the user input of “go to east” would be returned as [“move”, “east”].

And before passing the words list, the parser will check whether the list is empty or not.

```
//method to parse/determine the command input by the user
public static List<String> parseInput(String userInput){

    //converting the user input to lower case
    String command = userInput.toLowerCase();

    //boolean values to see if a specific command word has been found
    boolean foundExamineWords = false;
    boolean foundMoveWords = false;
    boolean foundTakeWords = false;
    boolean foundDropWords = false;


    //storing the user input into an ArrayList
    List<String> words = new ArrayList<>(Arrays.asList(command.split(regex: " ")));

    //lists containing possible words for specific in-game commands or possible inputs
    List<String> examineActions = new ArrayList<>(Arrays.asList(...a: "look", "check", "inspect", "examine", "study"));
    List<String> moveActions = new ArrayList<>(Arrays.asList(...a: "move", "go", "walk", "travel"));
    List<String> takeActions = new ArrayList<>(Arrays.asList(...a: "take", "grab", "nab"));
    List<String> quitActions = new ArrayList<>(Arrays.asList(...a: "quit", "exit", "q"));
    List<String> directions = new ArrayList<>(Arrays.asList(...a: "north", "n", "south", "s", "east", "e", "west", "w"));
    List<String> yesActions = new ArrayList<>(Arrays.asList(...a: "yes", "correct", "y", "yeah", "yea", "yep", "mhm", "right", "true"));
    List<String> noActions = new ArrayList<>(Arrays.asList(...a: "no", "n", "nope", "wrong", "nuhuh", "nah", "incorrect", "false"));

    int remainingWordsindex = 0;
    String remainingWords;
    List<String> result = new ArrayList<>();

    if(words.size() > 0){
```

❖ Examine Words

The examine words section of the method focuses on parsing words that are used to examine or take a closer look at objects, rooms, and NPCs.

It also handles words used to check the player's inventory.

```
//EXAMINE
//Parsing words for the examine command
if(words.get(index: 0).equals(anObject: "look") && words.get(index: 1).equals(anObject: "at")){
    remainingWordsindex = 2;
    foundExamineWords = true;

}
else if (examineActions.contains(words.get(index: 0))) {
    remainingWordsindex = 1;
    foundExamineWords = true;

}

if(foundExamineWords == true){
    if (words.size() > remainingWordsindex){
        remainingWords = "";
        for(int i = remainingWordsindex; i < words.size(); i++){
            remainingWords += words.get(i);

            //for objects/nouns with more than one word (e.g. book of spells)
            if (i < words.size() - 1){
                remainingWords += " ";
            }
        }
        command = "examine";
        result.add(command);
        result.add(remainingWords);
        return result;
    }
}

//Parsing words for checking inventory without examine command
if(words.size()==1 && (words.get(index: 0).equals(anObject: "inventory") || words.get(index: 0).equals(anObject: "i"))){
    remainingWordsindex = 0;
    command = "examine";
    result.add(command);
    result.add(e: "inventory");
    return result;
}
```

❖ Movement Words

The movement words section of the method focuses on parsing words that are used to move the player character from room to room in the map.

Movement Words continued:

```
//MOVE
//Parsing the words for the move command
//Parsing words for one word commands for moving
if(words.size()==1 && directions.contains(words.get(index: 0))){
    remainingWordsindex = 0;
    foundMoveWords = true;
}

if(moveActions.contains(words.get(index: 0)) && words.get(index: 1).equals(anObject: "to")){
    remainingWordsindex = 2;
    foundMoveWords = true;

}
else if (moveActions.contains(words.get(index: 0))) {
    remainingWordsindex = 1;
    foundMoveWords = true;

}


if(foundMoveWords == true){
    if (words.size() > remainingWordsindex){
        remainingWords = "";
        for(int i = remainingWordsindex; i < words.size(); i++){
            remainingWords += words.get(i);

            //for objects/nouns with more than one word (e.g. book of spells)
            if (i < words.size() - 1){
                remainingWords += " ";
            }

        }
        command = "move";
        result.add(command);
        result.add(remainingWords);
        return result;
    }
}
```

❖ Take Words

The take words section of the method focuses on parsing words that are used to take or pick up items from the environment.

```
//TAKE
//Parsing the words for the take command
if (takeActions.contains(words.get(index: 0))) {
    remainingWordsindex = 1;
    foundTakeWords = true;

}
if(words.get(index: 0).equals(anObject: "pick") && words.get(index: 1).equals(anObject: "up")){
    remainingWordsindex = 2;
    foundTakeWords = true;

}

if(foundTakeWords == true){
    if (words.size() > remainingWordsindex){
        remainingWords = "";
        for(int i = remainingWordsindex; i < words.size(); i++){
            remainingWords += words.get(i);
            //for objects/nouns with more than one word (e.g. book of spells)
            if (i < words.size() - 1){
                remainingWords += " ";
            }

        }
        command = "take";
        result.add(command);
        result.add(remainingWords);
        return result;
    }
}
```

❖ Drop Words

The drop words section of the method focuses on parsing words that are used to drop items from the player's inventory.

```
//DROP
//Parsing the words for the drop command
if (words.get(index: 0).equals(anObject: "drop")) {
    remainingWordsindex = 1;
    foundDropWords = true;

}
if((words.get(index: 0).equals(anObject: "set") || words.get(index: 0).equals(anObject: "put")) && words.get(index: 1).equals(anObject: "down")){
    remainingWordsindex = 2;
    foundDropWords = true;

}

if(foundDropWords == true){
    if (words.size() > remainingWordsindex){
        remainingWords = "";
        for(int i = remainingWordsindex; i < words.size(); i++){
            remainingWords += words.get(i);
            //for objects/nouns with more than one word (e.g. book of spells)
            if (i < words.size() - 1){
                remainingWords += " ";
            }
        }
        command = "drop";
        result.add(command);
        result.add(remainingWords);
        return result;
    }
}
```

❖ Yes and No Words

The yes and no section of the method focuses on parsing words that are used to confirm or deny a prompt (like in saving or loading a character).

```
//YES
//Parsing the words for the yes command
if (yesActions.contains(words.get(index: 0))) {

    command = "yes";
    result.add(command);
    return result;
}
```

```
//NO
//Parsing the words for the no command
if (noActions.contains(words.get(index: 0))) {

    command = "no";
    result.add(command);
    return result;
}
```

❖ Save, Load, and Delete Words

The save, load, and delete words section of the method focuses on parsing words that are used to save, load, and delete save files.

```
//SAVING
//Parsing the words for the save game
//save game, save progress, save
if(words.size() == 1 && (words.get(index: 0).equals(anObject: "save"))){
    command = "save";
    result.add(command);
    return result;
}

if(words.get(index: 0).equals(anObject: "save") && (words.get(index: 1).equals(anObject: "game") ||
                                                    words.get(index: 1).equals(anObject: "progress"))){
    command = "save";
    result.add(command);
    return result;
}


//LOADING
//Parsing the words for the load game command
//load game, load progress, continue save, load
if(words.size() == 1 && (words.get(index: 0).equals(anObject: "load"))){
    command = "load";
    result.add(command);
    return result;
}

if(words.size() == 2 && words.get(index: 0).equals(anObject: "load") && (words.get(index: 1).equals(anObject: "progress") ||
                                                                         words.get(index: 1).equals(anObject: "game") ||
                                                                         words.get(index: 1).equals(anObject: "save"))){
    command = "load";
    result.add(command);
    return result;
}

if(words.size() == 2 &&  words.get(index: 0).equals(anObject: "continue") && (words.get(index: 1).equals(anObject: "progress") ||
                                                                              words.get(index: 1).equals(anObject: "save") ||
                                                                              words.get(index: 1).equals(anObject: "game"))){
    command = "load";
    result.add(command);
    return result;
}
```

```
//DELETING
if(words.size() == 1 && (words.get(index: 0).equals(anObject: "delete"))){
    command = "delete";
    result.add(command);
    return result;
}

if(words.size() == 2 && words.get(index: 0).equals(anObject: "delete") && (words.get(index: 1).equals(anObject: "progress") ||
                                                                            words.get(index: 1).equals(anObject: "game") ||
                                                                            words.get(index: 1).equals(anObject: "save"))){
    command = "delete";
    result.add(command);
    return result;
}
```

❖ Quit Words

The quit words section focuses on parsing words that are used to quit or exit the game.

```
//QUIT
//Parsing the words for the quit command
if(quitActions.contains(words.get(index: 0))){
    command = "quit";
    result.add(command);
    return result;

}
```

❖ About and Commands Words

The about and commands words section of the method focuses on parsing words that are used to show the about and commands text for the player.

```
        //ABOUT and COMMANDS
        if(words.size() == 1 && words.get(index: 0).equals(anObject: "commands")){
            command = "commands";
            result.add(command);
            return result;
        }

        if(words.size() == 1 && words.get(index: 0).equals(anObject: "about")){
            command = "about";
            result.add(command);
            return result;
        }

        if(words.size() == 1 && words.get(index: 0).equals(anObject: "help")){
            command = "about";
            result.add(command);
            return result;
        }
    }

    return result;
}
```

That finishes up the parseInput method, now continuing with the other methods:

- **isNumberChoiceValid**, a method to check if the input String is a valid number for dialogue options. The possible number choices are 1, 2, and 3.

```
//method to check whether the inputted String is a valid number choice (for dialogue choices)
public static boolean isNumberChoiceValid(String input){
    //listing the possible number inputs for dialogue choices (at max 3)
    List<String> possibleChoices = new ArrayList<>(Arrays.asList(...a: "1","2","3"));
    if(!possibleChoices.contains(input)){
        return false;
    }else{
        return true;
    }
}
```

- **isInputPositiveInteger**, a method to check if an input String is a positive int.

```
//method to check if inputted string is a positive integer
static boolean isInputPositiveInteger(String input) {
    String regex = "[0-9]+";
    return input.matches(regex);
}
```

- **processInput**, a method to parse an input String fully. It returns a String list containing the result of the parsing process.

  The method first filters out special characters and punctuation. It then checks if the input is a positive integer. If it's found that the input String isn't a positive integer, it will call the parseInput method, else it will return the input as is.

```
//method to fully process/parse the inputted string
//filters out delims, parses the command
public static List<String> processInput(String input){
    List<String> parsed = new ArrayList<>();
    input = FilterDelims(input).trim();

    if(!isInputPositiveInteger(input)){
        parsed = parseInput(input);
    }else{
        parsed.add(input);
        return parsed;
    }
    return parsed;
}
```

d. Direction Enum

**Included in:** globals package

Direction is an enumerated type that assists in movement in the AdventureGame class. A static and final int called NOEXIT  with a value of -1 is used to signify when a Room does not have an exit in a cardinal direction.

```
1   package globals;
2
3   //enumerated type to make reading the code easier
4   public enum Direction{
5       NORTH,
6       SOUTH,
7       EAST,
8       WEST;
9
10      public static final int NOEXIT = -1;
11  }
```

e. Thing Class

**Included in:** gameobjects package

The Thing class is a class that is used to create all other in-game objects (NPCs, items, Rooms). It implements java.io.Serializable, in order for the game to save its data properly.

It has the following private attributes:

- **name,** a String to store the Thing's name
- **description**, a String to store the Thing's description text
- **examination**, a String to store the Thing's examination text
- **isPickupable**, a boolean value to determine if a Thing is able to be taken
- **isKeyItem**, a boolean value to determine if a Thing is a key item
- **location**, an int to store its location on the map
- **aliases**, a String ArrayList to store the possible aliases/nicknames the Thing can have

There are 4 constructor methods, each to instantiate a different type of Thing object:

- A general Thing object without any aliases
- The player character
- Actors other than the player and NPCs
- In-game items

```java
public class Thing implements java.io.Serializable{
    //Thing object that will be the superclass to all other game objects

    private String name, description, examination;
    private boolean isPickupable, isKeyItem;
    private int location;
    private ArrayList<String> aliases = new ArrayList<>();

    //constructor methods
    public Thing(String aName, String aDescription, String anExamination){
        this.name = aName;
        this.description = aDescription;
        this.examination = anExamination;
    }

    //constructor for the player character
    public Thing(String aName, String aDescription, String anExamination, ArrayList<String> aliases){
        this.aliases = aliases;
        this.name = aName;
        this.description = aDescription;
        this.examination = anExamination;
    }

    //constructor for NPCs
    public Thing(String aName, String aDescription, String anExamination, int aLocation, ArrayList<String> aliases){
        this.aliases = aliases;
        this.name = aName;
        this.description = aDescription;
        this.examination = anExamination;
        this.location = aLocation;
    }

    //constructor for Items
    public Thing(String aName, String aDescription, int aLocation, ArrayList<String> aliases, boolean isPickupable, boolean isKeyItem){
        this.aliases = aliases;
        this.name = aName;
        this.description = aDescription;
        this.isPickupable = isPickupable;
        this.isKeyItem = isKeyItem;
        this.location = aLocation;
    }
```

The Thing class has setter and getter methods for each attribute.

```java
//setters
//setting the Thing name
public void setName(String name){
    this.name = name;
}
//setting the Thing's location on the map
public void setLocation(int aLocation){
    this.location = aLocation;
}
//setting the Thing's description text
public void setDescription(String description){
    this.description = description;
}
//setting the Thing's examination text
public void setExamination(String examination){
    this.examination = examination;
}
//adding the Thing's aliases
public void addAlias(String alias){
    this.aliases.add(alias);
}
//setting if a Thing is able to be taken
public void setThingPickupability(boolean isPickupable){
    this.isPickupable = isPickupable;
}
//setting if a Thing is a key item
public void setThingKey(boolean isKeyItem){
    this.isKeyItem = isKeyItem;
}
```

```java
//getters
//returning all the attributes
public String getName(){
    return this.name;
}

public String getDescription(){
    return this.description;
}

public String getExamination(){
    return this.examination;
}

public ArrayList<String> getAliases(){
    return this.aliases;
}

public boolean isThingPickupable(){
    return this.isPickupable;
}

public boolean isThingKey(){
    return isKeyItem;
}

public int getLocation(){
    return this.location;
}
```

### f. ThingHolder Class

**Included in:** gameobjects package

The ThingHolder class is an extension of the Thing class and is used to create the Actor and Room classes (both of which are like containers which hold Thing objects).

Its only attribute is an ArrayList containing Thing objects named **things.**

The ThingHolder class has the following constructors for these ThingHolder objects:

- A room
- The player character
- Actors other than the player/NPCs

The ThingHolder class has the setThings and getThings methods, to set a Thing ArrayList and to return it.

```
public class ThingHolder extends Thing{

    private ArrayList<Thing> things = new ArrayList<>();

    //constructor for rooms
    public ThingHolder(String aName, String aDescription, String anExamination, ArrayList<Thing> tl){
        super(aName, aDescription, anExamination);
        this.things = tl;
    }

    //constructor for the player
    public ThingHolder(String aName, String aDescription, String anExamination, ArrayList<String> aliases, ArrayList<Thing> tl){
        super(aName, aDescription, anExamination, aliases);
        this.things = tl;
    }

    //constructor for NPCs
    public ThingHolder(String aName, String aDescription, String anExamination,  int aLocation, ArrayList<String> aliases, ArrayList<Thing> tl){
        super(aName, aDescription, anExamination, aLocation, aliases);
        this.things = tl;
    }

    //setter
    public void setThings(ArrayList<Thing> things){
        this.things = things;
    }

    //getter
    public ArrayList<Thing> getThings(){
        return things;
    }
}
```

### g. Room Class

**Included in:** gameobjects package

The Room class is an extension of the ThingHolder class and is used to create the rooms in the game.

It has the following attributes, in addition to the inherited ThingHolder class attributes:

- **north**, an int value to store the Room index connected to **this Room's** north exit
- **south,** an int value to store the Room index connected to **this Room's** south exit
- **east,** an int value to store the Room index connected to **this Room's** east exit
- **west,** an int value to store the Room index connected to **this Room's** east exit
- **isDialogueFinished,** a boolean value to store if the **this Room's** dialogue interaction has been completed
- **roomNPCs,** an Actor ArrayList containing the actors inside **this Room**

The Room class has only one constructor.

```
public class Room extends ThingHolder{

    private int north, south, east, west;
    private boolean isDialogueFinished;
    private ArrayList<Actor> roomNPCs = new ArrayList<>();

    //constructor method for a room
    public Room(String aName, String aDescription, String anExamination, int n, int s, int e, int w, ArrayList<Thing> tl){
        super(aName, aDescription, anExamination, tl);
        this.north = n;
        this.south = s;
        this.east = e;
        this.west = w;
    }
```

The Room class has setter and getter methods for each of its attributes, with some extra methods like:

- **getNPC,** to return a specific NPC in a Room by passing their name as a String
- **getNPCNames,** to return all room NPC names in a String ArrayList
- **getIfDialogueIsFinished,** to return a Room's isDialogueFinished attribute

```
//setters
//directional exits
public void setNorth(int north){
    this.north = north;
}

public void setSouth(int south){
    this.south = north;
}

public void setEast(int east){
    this.east = east;
}

public void setWest(int west){
    this.west = west;
}

//setting the room NPCs as an ArrayList of actors
public void setRoomNPCs(ArrayList<Actor> roomNPCs){
    this.roomNPCs = roomNPCs;
}

//setting whether the specific dialogue is finished or not
public void setIfDialogueIsFinished(boolean check){
    this.isDialogueFinished = check;
}
```

```
//getters
//getting the directional exits
public int getNorth(){
    return this.north;
}

public int getSouth(){
    return this.south;
}

public int getEast(){
    return this.east;
}

public int getWest(){
    return this.west;
}
```

```java
//getting a specific room NPC by searching their name
public Actor getNPC(String aName){
    for(Actor npc : this.roomNPCs){
        if(aName.equals(npc.getName()) || npc.getAliases().contains(aName));
        return npc;
    }
    return null;
}

//getting all room NPC's names
public ArrayList<String> getNPCNames(){
    ArrayList<String> roomNPCNames = new ArrayList<>();
    for(Actor npc : this.roomNPCs){
        roomNPCNames.add(npc.getName().toLowerCase());
        for(String alias : npc.getAliases()){
            roomNPCNames.add(alias.toLowerCase());
        }
    }
    return roomNPCNames;
}

//getting whether a dialogue is finished or not
public boolean getIfDialogueIsFinished(){
    return this.isDialogueFinished;
}
```

h. Actor Class

**Included in:** gameobjects package

The Actor class is an extension of the ThingHolder class and is used to create the player character and NPCs in the game.

It has the following attributes, in addition to the inherited ThingHolder class attributes:

- **anger,** a BigDecimal value to store the Actor's base anger
- **fear,** a BigDecimal value to store the Actor's base fear
- **relation,** a BigDecimal value to store the Actor's base relation
- **happiness,** a BigDecimal value to store the Actor's base happiness
- **angerTendency,** a BigDecimal value to store the Actor's emotional tendency towards the anger attribute
- **fearTendency,** a BigDecimal value to store the Actor's emotional tendency towards the fear attribute

- **relationTendency,** a BigDecimal value to store the Actor's emotional tendency towards the relation attribute
- **happinessTendency,** a BigDecimal value to store the Actor's emotional tendency towards the happiness attribute
- **angerTrigger,** a BigDecimal value to store the Actor's trigger value for their anger special action
- **fearTrigger,** a BigDecimal value to store the Actor's trigger value for their fear special action
- **relationTrigger,** a BigDecimal value to store the Actor's trigger value for their relation special action
- **happinessTrigger,** a BigDecimal value to store the Actor's trigger value for their happiness special action
- **inDialogueResponses,** a String ArrayList to store the Actor's in-dialogue responses to a player action
- **inDialogueWith,** a String to store the name of the other Actor **this Actor** is currently in a dialogue with
- **room,** a Room object that shows which room the Actor is currently in
- **isInDialogue,** a boolean value to check if the Actor is currently in a dialogue interaction
- **roomVisits,** a HashMap of Rooms (key) and Integers (values) to store how many times the Actor has visited a specific room

The Actor class has the following constructors:

- The player character
- Actors other than the player/NPCs

```
public class Actor extends ThingHolder {

    private BigDecimal anger, fear, relation, happiness,
                       angerTendency, fearTendency, relationTendency, happinessTendency,
                       angerTrigger, fearTrigger, relationTrigger, happinessTrigger;

    private ArrayList<String> inDialogueResponses = new ArrayList<>();
    private String inDialogueWith;
    private Room room;
    private boolean isInDialogue;
    private Map<Room, Integer> roomVisits = new HashMap<Room, Integer>();

    //constructor for player
    public Actor(String aName, String aDescription, String anExamination, Room aRoom, ArrayList<String> aliases, ArrayList<Thing> tl, boolean isInDialogue, String inDialogueWith){
        super(aName, aDescription, anExamination, aliases, tl);
        this.room = aRoom;
        this.isInDialogue = isInDialogue;
        this.inDialogueWith = inDialogueWith;
    }

    //constructor for NPCs
    public Actor(String aName, String aDescription, String anExamination, Room aRoom, int location, ArrayList<String> aliases, ArrayList<Thing> tl,
                 BigDecimal anger, BigDecimal fear, BigDecimal relation, BigDecimal happiness,
                 BigDecimal angerTendency, BigDecimal fearTendency, BigDecimal relationTendency, BigDecimal happinessTendency,
                 ArrayList<String> inDialogueResponses, BigDecimal angerTrigger, BigDecimal fearTrigger, BigDecimal relationTrigger, BigDecimal happinessTrigger){

        super(aName, aDescription, anExamination, location, aliases, tl);
        this.room = aRoom;
        this.anger = anger;
        this.fear = fear;
        this.relation = relation;
        this.happiness = happiness;
        this.angerTendency = angerTendency;
        this.fearTendency = fearTendency;
        this.relationTendency = relationTendency;
        this.happinessTendency = happinessTendency;
        this.inDialogueResponses = inDialogueResponses;
        this.angerTrigger = angerTrigger;
        this.fearTrigger = fearTrigger;
        this.relationTrigger = relationTrigger;
        this.happinessTrigger = happinessTrigger;

    }
```

The Actor class has setter and getter methods for each of its attributes, as well as other methods such as:

- **calculateAnger,** to calculate and set the resulting anger after an attribute change
- **calculateFear,** to calculate and set the resulting fear after an attribute change
- **calculateRelation,** to calculate and set the resulting relation after an attribute change
- **calculateHappiness,** to calculate and set the resulting happiness after an attribute change
- **calculateAllAttributes,** to calculate and set all of the resulting attributes after attribute changes
- **addRoomAndVisits,** to add a Room and its visit value to the Actor's roomVisits HashMap
- **addVisits**, to increment the visit value of a Room in the roomVisits HashMap by 1
- **getExamineResponse,** to return index zero (examine response) of the DialogueResponses String ArrayList
- **getMoveResponse,** to return the index 1 (move response) of the DialogueResponses String ArrayList
- **getTakeResponse,** to return the index 2 (take response) of the DialogueResponses String ArrayList
- **getDropResponse,** to return the index 3 (drop response) of the DialogueResponses String ArrayList
- **getSpecificRoomVisits,** to return a specific Room's visit value in the roomVisits HashMap
- **printNPCCurrentAttributes,** to display the NPC's current attributes
- **toString**, overridden method to return the NPC name and description as a String

```
//setters
public void setRoom(Room aRoom){
    this.room = aRoom;
}
//setting the actor's attributes
public void setAnger(BigDecimal anger){
    this.anger = anger;
}
public void setFear(BigDecimal fear){
    this.fear = fear;
}
public void setRelation(BigDecimal relation){
    this.relation = relation;
}
public void setHappiness(BigDecimal happiness){
    this.happiness = happiness ;
}

//setting the actor's attribute tendencies (how prone are they to each attribute)
public void setAngerTendency(BigDecimal angerTendency){
    this.angerTendency = angerTendency;
}
public void setFearTendency(BigDecimal fearTendency){
    this.fearTendency = fearTendency;
}
public void setRelationTendency(BigDecimal relationTendency){
    this.relationTendency = relationTendency;
}
public void setHappinessTendency(BigDecimal happinessTendency){
    this.happinessTendency = happinessTendency ;
}
```

```
//setting the actor's special action trigger attributes
public void setAngerTrigger(BigDecimal angerTrigger){
    this.angerTrigger = angerTrigger;
}
public void setFearTrigger(BigDecimal fearTrigger){
    this.fearTrigger = fearTrigger;
}
public void setRelationTrigger(BigDecimal relationTrigger){
    this.relationTrigger = relationTrigger;
}
public void setHappinessTrigger(BigDecimal happinessTrigger){
    this.happinessTrigger = happinessTrigger;
}


//setting whether an actor is in a dialogue
public void setIfActorIsInDialogue(boolean isInDialogue){
    this.isInDialogue = isInDialogue;
}
//setting which other actor this actor is in a dialogue with
public void setWhoActorInDialogueWith(String aName){
    this.inDialogueWith = aName;
}
//adding new rooms and their visiting number to the rooms and visits hashmap
public void addRoomAndVisits(Room aRoom, int numVisits){
    this.roomVisits.put(aRoom, numVisits);
}
//add a visit (increment by 1) to the passed room's value
public void addVisit(Room aRoom){
    this.roomVisits.merge(aRoom, value: 1, Integer::sum);
}
```

```
//methods for adding to their attributes with a multiplier from their tendencies
public void calculateAnger(BigDecimal angerChange){
    if(!(angerChange.compareTo(BigDecimal.ZERO) == 0)){
        //checking if the value is positive or not
        if(this.angerTendency.compareTo(BigDecimal.ZERO) < 0){
            this.anger = this.anger.subtract(angerChange.multiply(getAngerTendency()));
        }else{
            this.anger = this.anger.add(angerChange.multiply(getAngerTendency()));
        }
    }
}
public void calculateFear(BigDecimal fearChange){
    //checking if the value is positive or not
    if(!(fearChange.compareTo(BigDecimal.ZERO) == 0)){
        if(this.fearTendency.compareTo(BigDecimal.ZERO) < 0){
            this.fear = this.fear.subtract(fearChange.multiply(getFearTendency()));
        }else{
            this.fear = this.fear.add(fearChange.multiply(getFearTendency()));
        }
    }
}
public void calculateRelation(BigDecimal relationChange){
    //checking if the value is positive or not
    if(!(relationChange.compareTo(BigDecimal.ZERO) == 0)){
        if(this.relationTendency.compareTo(BigDecimal.ZERO) < 0){
            this.relation = this.relation.subtract(relationChange.multiply(getRelationTendency()));
        }else{
            this.relation = this.relation.add(relationChange.multiply(getRelationTendency()));
        }
    }

}

public void calculateHappiness(BigDecimal happinessChange){
    //checking if the value is positive or not
    if(!(happinessChange.compareTo(BigDecimal.ZERO) == 0)){
        if(this.happinessTendency.compareTo(BigDecimal.ZERO) < 0){
            this.happiness = this.happiness.subtract(happinessChange.multiply(getHappinessTendency()));
        }else{
            this.happiness = this.happiness.add(happinessChange.multiply(getHappinessTendency()));
        }
    }
}
```

```
//for calculating all attribute changes
public void calculateAllAttributes(BigDecimal angerChange, BigDecimal fearChange,
                                   BigDecimal relationChange, BigDecimal happinessChange){
    calculateAnger(angerChange);
    calculateFear(fearChange);
    calculateRelation(relationChange);
    calculateHappiness(happinessChange);
}
```

```java
//getters
//getting the room the actor is in
public Room getRoom(){
    return this.room;
}

//getting the actor's attributes
public BigDecimal getAnger(){
    return this.anger;
}

public BigDecimal getFear(){
    return this.fear;
}

public BigDecimal getRelation(){
    return this.relation;
}

public BigDecimal getHappiness(){
    return this.happiness;
}

//getting the actor's attribute tendencies
public BigDecimal getAngerTendency() {
    return this.angerTendency;
}
public BigDecimal getFearTendency() {
    return this.fearTendency;
}
public BigDecimal getRelationTendency() {
    return this.relationTendency;
}

public BigDecimal getHappinessTendency(){
    return this.happinessTendency;
}
```

```java
//getting the actor's special action trigger attributes
public BigDecimal getAngerTrigger(){
    return this.angerTrigger;
}

public BigDecimal getFearTrigger(){
    return this.fearTrigger;
}

public BigDecimal getRelationTrigger(){
    return this.relationTrigger;
}

public BigDecimal getHappinessTrigger(){
    return this.happinessTrigger;
}

//getting whether an actor is currently in dialogue
public boolean isActorInDialogue(){
    return this.isInDialogue;
}

//getting the actor's various responses when in dialogue
public String getExamineResponse(){
    return this.inDialogueResponses.get(index: 0);
}

public String getMoveResponse(){
    return this.inDialogueResponses.get(index: 1);
}

public String getTakeResponse(){
    return this.inDialogueResponses.get(index: 2);
}

public String getDropResponse(){
    return this.inDialogueResponses.get(index: 3);
}
```

```java
//get which actor this actor is engaged in dialogue with
public String getWhoActorInDialogueWith(){
    return this.inDialogueWith;
}

//get how many times a specific room has been visited
public int getSpecificRoomVisits(Room aRoom){
    for (Room i : this.roomVisits.keySet()) {
        if(i.equals(aRoom)){
            return this.roomVisits.get(i);
        }
      }
    return 0;
}

//method to print the current NPC attributes
public void printNPCCurrentAttributes(){
    System.out.println("Anger\t\t: " + getAnger());
    System.out.println("Fear\t\t:  " + getFear());
    System.out.println("Relation\t:  " + getRelation());
    System.out.println("Happiness\t: " + getHappiness());
}

//toString method to return an NPC's profile, consisting of their name and description
@Override
public String toString(){
    return "Profile:\nName\t\t: " + this.getName() + "\nProfile\t\t: " + this.getDescription() + "\n";
}
```

### i. Dialogue Class

**Included in:** gameobjects package

The Dialogue class is used to store the attributes that a line of dialogue might have (specifically for lines that are dialogue options for the player). It also implements the java.io.Serializable interface.

It has the following attributes:

- **nextDialogueline,** a String ArrayList to store the next line of dialogue (numeric)
- **dialogue,** a String to store the text that will be displayed as the dialogue
- **angerChange,** a BigDecimal value to store the amount of change a dialogue action will do to an Actor's anger attribute
- **fearChange,** a BigDecimal value to store the amount of change a dialogue action will do to an Actor's fear attribute
- **relationChange,** a BigDecimal value to store the amount of change a dialogue action will do to an Actor's relation attribute
- **happinessChange,** a BigDecimal value to store the amount of change a dialogue action will do to an Actor's happiness attribute

It only has the default constructor.

```
public class Dialogue implements java.io.Serializable {

    private ArrayList<String> nextDialogueLine;
    private String dialogueText;
    private BigDecimal angerChange, fearChange, relationChange, happinessChange;

    public Dialogue(){}
```

It has setter and getter methods for each of its attributes.

```
//setters
//setting the next dialogue line that is linked to the current one
public void setNextDialogueLine(ArrayList<String> nextDialogueLine){
    this.nextDialogueLine = nextDialogueLine;
}

//setting the actual text of dialogue as a String
public void setDialogueText(String dialogue){
    this.dialogueText = dialogue;
}

//setting the attribute changes for the dialogue
//if the dialogue is found to be a dialogue option
public void setAngerChange(BigDecimal angerChange){
    this.angerChange = angerChange;
}

public void setFearChange(BigDecimal fearChange){
    this.fearChange = fearChange;
}

public void setRelationChange(BigDecimal relationChange){
    this.relationChange = relationChange;
}

public void setHappinessChange(BigDecimal happinessChange){
    this.happinessChange = happinessChange;
}
```

```
//getters
//returning the next linked dialogue lines as a String ArrayList
public ArrayList<String> getNextDialogueLine(){
    return this.nextDialogueLine;
}

//returning the first dialogue from the list of the next linked dialogue as an int
//the int represents which line in the text file the dialogue will be read from
public int getFirstNextDialogue(){
    int firstnextDialogueLine = Integer.parseInt(this.nextDialogueLine.get(index: 0));
    return firstnextDialogueLine;
}

//returning the dialogue text as a String
public String getDialogueText(){
    return this.dialogueText;
}

//returning the attribute changes as a BigDecimal
//if the dialogue is found to be a dialogue option
public BigDecimal getAngerChange(){
    return this.angerChange;
}

public BigDecimal getFearChange(){
    return this.fearChange;
}

public BigDecimal getRelationChange(){
    return this.relationChange;
}

public BigDecimal getHappinessChange(){
    return this.happinessChange;
}
```

j. AdventureGame Class

**Included in:** game package

The AdventureGame class is a class that implements the java.io.Serializable interface and contains the main logic and code Zwielicht runs on. I'll be explaining it section by section to make the information more digestible.

1. Data Section

The AdventureGame class has the following attributes:

- **map**, a Room ArrayList to store the locations
- **allItems**, a Thing ArrayList to store all the items
- **roomNames**, a String ArrayList to store the names of all locations
- **allItemNames**, a String ArrayList to store the names of all items
- **allNPCNames**, a String ArrayList to store the names of all NPCs
- **playerInventory**, a Thing ArrayList to store the player's items
- **playerAliases,** a String ArrayList to store the player's aliases
- **player,** an Actor object for the player character
- **nextDialogueList,** an Integer ArrayList to store the possible upcoming dialogue lines
- **listOfChoices**, a Dialogue ArrayList to store the Dialogue objects when the player is faced with branching dialogue
- **playerVisits,** a HashMap of Rooms (key) and Integers (value) to keep track of how many times the player has visited a room
- **isSpecialActionTriggered,** an int to store if an NPC special actions have been triggered and also determines what type of special action it was.
  The corresponding int values are:
  - 0, not triggered
  - 1, anger special action
  - 2, fear special action
  - 3, relation special action
  - 4, happiness special action
- **timesSpecialActionWasTriggered,** an int to store how many times special actions have been triggered.
- **commands,** a String list of recognized commands/verbs
- **directions,** a String list of recognized directions
- **miscNouns**, a String list of miscellaneous nouns that are valid, such as "inventory" and "around"

```java
public class AdventureGame implements java.io.Serializable{

    //Creating the needed assets
    private ArrayList<Room> map;
    private ArrayList<Thing> allItems;
    private ArrayList<String> roomNames;
    private ArrayList<String> allItemNames;
    private ArrayList<String> allNPCNames;
    private ArrayList<Thing> playerInventory;
    private ArrayList<String> playerAliases;
    private Actor player;

    //ArrayLists for dialogue interaction (for the upcoming dialogue line)
    private ArrayList<Integer> nextDialogueList = new ArrayList<>();
    //for the list of dialogue options the player can choose
    private ArrayList<Dialogue> listOfChoices = new ArrayList<>();

    //hashmap of the rooms and their visit values
    private Map<Room, Integer> playerVisits = new HashMap<>();

    //private int values for checking if a special action has been triggered
    //and also the variant of the special action (anger, fear, relation, or happiness)
    private int isSpecialActionTriggered = 0;

    //private int value for checking the number of times a special action has been triggered
    private int timesSpecialActionWasTriggered = 0;

    //list of recognized commands
    private List<String> commands = new ArrayList<>(Arrays.asList(...a: "examine", "move", "talk", "take", "drop", "use", "yes", "no", "quit"));
    private List<String> directions = new ArrayList<>(Arrays.asList(...a: "north", "south", "east", "west"));
    private List<String> miscNouns = new ArrayList<>(Arrays.asList(...a: "inventory", "i", "around"));
```

2. Constructor Method

The AdventureGame class only has 1 constructor method. It's used to create the map, NPCs, items, and player character by reading their corresponding text files.

```java
//constructor
public AdventureGame(){
    //creating the game map + a list of all scene names by reading the Rooms file
    this.map = ReadFile.createRooms(itemsPathName: ".\\..\\gamefiles\\Items.txt", roomsPathName: ".\\..\\gamefiles\\Rooms.txt", charactersPathName: ".\\..\\gamefiles\\Characters.txt");
    this.roomNames = getAllRoomNames();

    //getting the list of all NPC names
    this.allNPCNames = getAllNPCNames();

    //creating a list of all items + all item names by reading the Items file
    this.allItems = ReadFile.createItems(pathName: ".\\..\\gamefiles\\Items.txt");
    this.allItemNames = HelpMethods.allThingNamesList(allItems);

    //creating the player inventory
    playerInventory = new ArrayList<Thing>();

    //setting the scenes and their visit values in a hashmap
    setRoomVisitMap();

    //player aliases
    playerAliases = new ArrayList<>(Arrays.asList(...a: "me", "myself"));

    //creating the player character
    player = new Actor(aName: "Vega", aDescription: "Not worth getting into my backstory.", anExamination: "Really, nothing noteworthy here.",
                       map.get(index: 0), playerAliases, playerInventory, isInDialogue: false, inDialogueWith: "");

}
```

3. About/Help Section

This section shows the methods used to show the about and commands text using the printFile method from the ReadFile class.

```
//ABOUT/HELP SCREENS
//reading and printing the about file
private void showAbout(){
    ReadFile.printFile(pathName: ".\\..\\gamefiles\\About.txt");
}

//reading and printing the commands file
private void showCommands(){
    ReadFile.printFile(pathName: ".\\..\\gamefiles\\Commands.txt");

}
```

4. Dialogue Section

This section shows how the game handles a dialogue interaction, involving methods such as:

- **startDialogue,** a method to signal to the program that a dialogue interaction started
- **endDialogue,** a method to signal to the program that a dialogue interaction ended

```
//DIALOGUE----------------------------------------------------------------------------
//method for starting a dialogue interaction
public void startDialogue(String actorName, String fileName){
    //signals the player is in an active dialogue
    getPlayer().setIfActorIsInDialogue(isInDialogue: true);
    //keeps processing until the dialogue is finished
    while(getPlayer().isActorInDialogue()){
        //setting the intended actor's name as the player's inDialogueWith attribute
        getPlayer().setWhoActorInDialogueWith(actorName);
        //printing the active dialogue NPC's description
        System.out.println(getActiveDialogueNPC().getDescription());
        //calling the method to process the dialogue interaction
        processActorScene(count: 1, fileName, getActiveDialogueNPC());
    }
}

//method for ending a dialogue interaction
public void endDialogue(){
    //signals the player has ended a dialogue interaction
    getPlayer().setIfActorIsInDialogue(isInDialogue: false);
    //signals the room's dialogue interaction is completed
    getActiveRoom().setIfDialogueIsFinished(check: true);
}
```

- **addDialogueAttributeChanges,** to set the attribute changes of a Dialogue object

```
//method for setting a dialogue's possible actor attribute changes
private void addDialogueAttributeChanges(Dialogue aDialogue, ArrayList<String> dialogueLine){
    //getting the attribute changes by splitting the dialogue line
    ArrayList<String> attributeChanges = ReadFile.splitLineByCommas(dialogueLine, position: 2);
    //setting the respective attribute changes to a variable
    BigDecimal angerChange = new BigDecimal(attributeChanges.get(index: 0));
    BigDecimal fearChange = new BigDecimal(attributeChanges.get(index: 1));
    BigDecimal relationChange = new BigDecimal(attributeChanges.get(index: 2));
    BigDecimal happinessChange = new BigDecimal(attributeChanges.get(index: 3));
    //setting the dialogue's respective attribute changes
    aDialogue.setAngerChange(angerChange);
    aDialogue.setFearChange(fearChange);
    aDialogue.setRelationChange(relationChange);
    aDialogue.setHappinessChange(happinessChange);

}
```

- **processAttributeChanges,** to calculate all attribute changes made to an NPC

```
//method for processing the actor attribute changes
private void processAttributeChange(int dialogueOptionIndex){
    getActiveDialogueNPC().calculateAllAttributes(getADialogueOption(dialogueOptionIndex).getAngerChange(),
                                                  getADialogueOption(dialogueOptionIndex).getFearChange(),
                                                  getADialogueOption(dialogueOptionIndex).getRelationChange(),
                                                  getADialogueOption(dialogueOptionIndex).getHappinessChange());
}
```

- **getNextDialogueList,** to return the ArrayList containing the next linked dialogues

```
//method for returning the list containing the next lines of dialogue
public ArrayList<Integer> getNextDialogueList(){
    return this.nextDialogueList;
}
```

- **getDialogues,** a recursive method to return the list of choices (dialogue options) and print the dialogue text for the player

```
//recursive method for parsing and getting the dialogue
public ArrayList<Dialogue> getDialogues(String aLine) throws NumberFormatException, IOException{
    //splitting the dialogue line by the @ symbol
    ArrayList<String> splitDialogue = ReadFile.splitLineByAtSymbol(aLine);
    //splitting the last index of the line to get the linked dialogue
    ArrayList<String> nextDialogueLine = ReadFile.splitLineByCommas(splitDialogue, -1);
    //creating a new dialogue object
    Dialogue myDialogue = new Dialogue();
    //setting the linked dialogue to the nextDialogueLine attribute
    myDialogue.setNextDialogueLine(nextDialogueLine);
    //setting the dialogue text to be shown to the dialogueText attribute
    myDialogue.setDialogueText(splitDialogue.get(index: 1));

    //checking if the next linked dialogue is the last one (has a value of -1)
    if(myDialogue.getFirstNextDialogue() != -1){
        //checking if the dialogue is a text or an option
        if(splitDialogue.get(index: 0).equals(anObject: "text")){
            //clearing the nextDialogueList
            nextDialogueList.clear();
            //printing the dialogue text
            System.out.println(myDialogue.getDialogueText());
            System.out.println();
            //finding the next linked dialogue line in the text file
            for(String number : myDialogue.getNextDialogueLine()){
                String nextLine = ReadFile.getSpecificLine(Integer.parseInt(number.trim()), fileName: ".\\..\\gamefiles\\Scene.txt");
                getDialogues(nextLine);//calling the method again
            }
        }
        else{
            //if it's a dialogue option, adds the possible attribute changes to the dialogue object
            addDialogueAttributeChanges(myDialogue, splitDialogue);
            //adding the next linked dialogue
            nextDialogueList.add(myDialogue.getFirstNextDialogue());
            //printing out the dialogue option with their number choice
            System.out.println(nextDialogueList.size() + ". " + myDialogue.getDialogueText());
            //adding the dialogue object to the list of choices
            listOfChoices.add(myDialogue);
        }
    }else{
        //if the next linked dialogue isn't the last, clears the nextDialogueList
        nextDialogueList.clear();
        //prints out the dialogue text
        System.out.println(myDialogue.getDialogueText());
    }
    return listOfChoices;

}
```

- **processActorScene**, a recursive method that will keep asking for player input until the dialogue interaction is ended.

    - It prints out the dialogue text and options using the **getDialogues** method
    - Then checks if an NPC special action has been triggered using the **checkForActorSpecialActions** method
        - If it's triggered and it was the first time, it will call the **processSpecialActions** method
        - If it's not found, or has been triggered more than once, it will then continue to the next line of code
    - Next, it will check if the nextDialogueList is empty (last line of dialogue).
        - If it is, then it will end the dialogue and notify the player
        - If it isn't, then it will ask for user input (to make a choice in the dialogue) and when it gets a valid input, it will run the command associated.

```
//recursive method to process a dialogue interaction
public void processActorScene(int count, String fileName, Actor anActor){
    try{
        List<String> outputs = new ArrayList<>();
        String output;
        BufferedReader userReader = new BufferedReader(new InputStreamReader(System.in));
        String nextLine = ReadFile.getSpecificLine(count, fileName);
        listOfChoices = getDialogues(nextLine);
        System.out.println();
        //checking if a special action was triggered
        //a special action can only be triggered once
        isSpecialActionTriggered = checkForActorSpecialActions(anActor); //updates the isSpecialActionTriggered var if triggered
        //if a special action hasn't been triggered, checks for it
        if(timesSpecialActionWasTriggered == 0){
            switch(isSpecialActionTriggered){
                //anger special action
                case 1:
                    processActorSpecialActions(typeOfSpecialAction: 1, anActor);
                    timesSpecialActionWasTriggered = 1;
                    break;
                //fear special action
                case 2:
                    processActorSpecialActions(typeOfSpecialAction: 2, anActor);
                    timesSpecialActionWasTriggered = 1;
                    break;
                //relation special action
                case 3:
                    processActorSpecialActions(typeOfSpecialAction: 3, anActor);
                    timesSpecialActionWasTriggered = 1;
                    break;
                //
                case 4:
                    processActorSpecialActions(typeOfSpecialAction: 4, anActor);
                    timesSpecialActionWasTriggered = 1;
                    break;
                default:
                    break;
            }
        }
```

```
        //if the list of the next dialogue is empty, ends the dialogue
        if(this.getNextDialogueList().isEmpty() || !getPlayer().isActorInDialogue()){
            endDialogue();
            System.out.println("This is the end of the " + HelpMethods.capitalizeString(anActor.getName()) +
                               " Dialogue.\nYou are now free to roam around the map. Have fun!");
            timesSpecialActionWasTriggered = 0;
        }
        //next dialogue is still found, continues the interaction
        else{
            //asking for user input
            System.out.print(s: "> ");
            String choice = userReader.readLine();
            outputs = TextParser.processInput(choice); //parsing the user input
            output = runCommand(outputs);  //running the command
            if(!outputs.isEmpty()){  //if the initial outputs isn't empty, continues to run
                //checks if the input was a numeric choice (dialogue choice)
                if(HelpMethods.isNumeric(output)){
                    count = Integer.parseInt(output);
                    processActorScene(count, fileName, anActor);
                }
                //handling input that's not a dialogue choice (movement, examining, etc)
                else{
                    System.out.println(output);
                    processActorScene(count, fileName, anActor);
                }
            }else{ //handling empty/unrecognized inputs
                System.out.println(output);
                processActorScene(count, fileName, anActor);
            }
        }
    } catch (IOException e){
        System.out.println(x: "File could not be accessed, try again!");
    }
}
```

5. NPC Special Actions

This section will focus on NPC special actions and processing them.

- **checkForActorSpecialActions,** checks on whether an Actor's special action trigger has been reached by comparing the Actor's current attributes to its respective trigger values. If a trigger value was reached, it returns an int with that attribute's specific value (anger = 1, fear = 2, relation = 3, happiness = 4). If no trigger values were reached, it returns an int with a value of 0.

```
//method for processing what happens if you reach Edelmar's special action trigger values
private void processActorSpecialActions(int typeOfSpecialAction, Actor anActor){
    String actorName = anActor.getName();
    //stating to the player that a special action has been triggered
    String msg = "One of " + HelpMethods.capitalizeString(actorName) + " Special Actions has been triggered.\n\n";
    //anger trigger value reached
    if(typeOfSpecialAction == 1){
        endDialogue();
        msg += ReadFile.fileToString(".\\..\\gamefiles\\NPCs\\"+ actorName + "\\AngerSA.txt");
        moveActorTo(getPlayer(), this.map.get(index: 0));
    }
    //fear trigger value reached
    else if(typeOfSpecialAction == 2){
        //el talks about your love life, and how it's been deteriorating with each new paramour
        msg += ReadFile.fileToString(".\\..\\gamefiles\\NPCs\\"+ actorName + "\\FearSA.txt");

    }//relation trigger value reached
    else if(typeOfSpecialAction == 3){
        //el gifts you an old dagger (deer catcher), probably owned by his family
        msg += ReadFile.fileToString(".\\..\\gamefiles\\NPCs\\"+ actorName + "\\RelationSA.txt");
        transferItem(HelpMethods.thisThing(aName: "hunting dagger", allItems), getActiveDialogueNPC().getThings(), getPlayer().getThings());
    }else if(typeOfSpecialAction == 4){
        //el asks you about esther
        msg += ReadFile.fileToString(".\\..\\gamefiles\\NPCs\\"+ actorName + "\\HappinessSA.txt");
    }
    System.out.println(msg);
}
```

- **processActorSpecialActions**, processes the special action events by determining which type was triggered and reading its corresponding text file. Since there's only one NPC in the game as of now, I only gave a few possible special actions. If there were more NPCs, I'd implement a switch case to determine which NPC's special actions were triggered.

```
//method for checking if an NPC's attribute trigger has been reached
public int checkForActorSpecialActions(Actor anActor){
    //comparing an actor's current attributes to their respective attribute triggers
    int compareAnger = anActor.getAngerTrigger().compareTo(anActor.getAnger());
    int compareFear = anActor.getFearTrigger().compareTo(anActor.getFear());
    int compareRelation = anActor.getRelationTrigger().compareTo(anActor.getRelation());
    int compareHappiness = anActor.getHappinessTrigger().compareTo(anActor.getHappiness());

    //check if any of the current atrributes qualify for a special action

    if(compareAnger == 0|| compareAnger == -1){ //checks if the current anger >= angerTrigger
        isSpecialActionTriggered = 1;
    }else if(compareFear == 0 || compareFear == 1){ //checks if current fear <= fearTrigger
        isSpecialActionTriggered = 2;
    }
    else if(compareRelation == 0 || compareRelation == -1){ //checks if current relation >= relationTrigger
        isSpecialActionTriggered = 3;
    }
    else if(compareHappiness == 0 || compareHappiness == -1){//checks if current happiness >= happinessTrigger
        isSpecialActionTriggered = 4;
    }
    else{ //if no attributes qualify, return 0
        isSpecialActionTriggered = 0;
    }
    return isSpecialActionTriggered;
}
```

6. Examining Items

This section shows how the program processes the "examine" command.

With the **examineItem** method, the item is compared to a list of items/NPCs in the player's current location. If the player is currently in a dialogue interaction, the NPC's anger attribute will be increased by 1 and a response from the NPC will also be added.

It compares the inputted itemName with:

- Items in the player's inventory
- Items in the environment
- Player and NPC names and aliases
- The word "inventory" and "i" for checking player inventory
- The word "around" to examine a location closer

After that, it checks whether it has found a roomItem, anInventoryItem, or an item in an NPC's inventory.

```
//EXAMINING----------------------------------------------------------------------
public String examineItem(String itemName){
    String msg = "";
    Thing aRoomItem = HelpMethods.thisThing(itemName, getActiveRoom().getThings());
    Thing anInventoryItem = HelpMethods.thisThing(itemName, getPlayer().getThings());

    //flags for whether a roomItem or an InventoryItem is found
    boolean roomItemFound = true;
    boolean inventoryItemFound = true;

    //checking inventory
    if((itemName.equals(anObject: "inventory") || itemName.equals(anObject: "i"))){ // check the i command
        showInventory(getPlayer());
        return msg;
    }

    //if the player enters an itemName that equates to the player character themselves
    else if(playerAliases.contains(itemName) || itemName.equals(getPlayer().getName())){
        msg = getPlayer().getExamination();
        return msg;
    }

    //checking if the itemName matches with any NPCs in the game
    else if(allNPCNames.contains(itemName)){
        if(getPlayer().isActorInDialogue()){
            msg += getActiveDialogueNPC().getExamineResponse() + "\n\n";
            msg += getActiveDialogueNPC().getExamination();
            return msg;
        }

        if(getActiveRoom().getNPC(itemName).equals(getActiveDialogueNPC())){
            msg = getActiveRoom().getNPC(itemName).getExamination();
        }else{
            msg = "They're not in your current area. How would you examine them?\n";
        }
        return msg;
    }

    //examining the active scene
    else if(itemName.equals(anObject: "around")){
        msg = getActiveRoom().getExamination();
        return msg;
    }
```

```
    //examining the active scene
    else if(itemName.equals(anObject: "around")){
        msg = getActiveRoom().getExamination();
        return msg;
    }

    //checking if the desired item is an inventory item
    if(anInventoryItem == null){
        msg = "There's nothing named " + itemName + " in your inventory.\n";
        inventoryItemFound = false;
    }

    //checking if the desired item is an item in the room
    if(aRoomItem == null){
        msg = "There's nothing named " + itemName + " in this location.\n";
        roomItemFound = false;

    }

    //if the item in question is in an NPC's inventory
    if(getActiveDialogueNPC().getThings().contains(HelpMethods.thisThing(itemName, getActiveDialogueNPC().getThings()))){
        msg = "You can't really get a closer look.\n";

    }

    //found an item in the inventory
    else if (!roomItemFound && inventoryItemFound){
        msg = anInventoryItem.getDescription();
    }

    //found an item in the room
    else if(roomItemFound && !inventoryItemFound){
        msg = aRoomItem.getDescription();
    }

    //adding an extra NPC response if the player is in dialogue
    if(getPlayer().isActorInDialogue()){
        getActiveDialogueNPC().calculateAnger(BigDecimal.ONE);
        msg += getActiveDialogueNPC().getExamineResponse() + "\n\n" ;
    }



    return msg;
}
//EXAMINE END----------------------------------------------------------------------
```

7. Taking and Dropping Items

This section will cover the program processes “take” and “drop” commands.

With the **transferItem** method, an item can be transferred from one list to another, this is how taking and dropping fundamentally works in the game.

The **showInventory** method is used to show a list of the player's inventory items

The **take**Item method will check if the named item is found in the room and if it's able to be picked up. If it passes both checks, the item will be transferred into the player's inventory.

```
//TAKING AND DROPPING-----------------------------------------------------------------

//method to transfer items from one list to another
private void transferItem(Thing aThing, ArrayList<Thing> fromList, ArrayList<Thing> toList){
    fromList.remove(aThing);
    toList.add(aThing);
}

//method to show a character's inventory
private void showInventory(Actor anActor){
    System.out.println(x: "-------------------------------------------------INVENTORY-------------------------------------------------\n");
    System.out.println(HelpMethods.describeThings(anActor.getThings()));
    System.out.println(x: "\n-----------------------------------------------------------------------------------------------------------\n");
}

//method to take an item and put it in the player's inventory
public String takeItem(String itemName){
    String msg = "";
    boolean takeSuccess = false; //checks if the take action was successful
    Thing t = HelpMethods.thisThing(itemName, getActiveRoom().getThings());
    if (itemName.equals(anObject: "")){ //checks if the itemName is an empty string
        itemName = "nothing";
    }if(t == null){ //checks if the itemName doesn't exist
        msg = "There's nothing named " + itemName + " in here.\n";
    }else{
        if(t.isThingPickupable() == true){ //checks if the item is able to be taken
            transferItem(t, player.getRoom().getThings(), player.getThings());
            msg =  HelpMethods.capitalizeString(itemName) + " has been taken.";
            takeSuccess = true;

        }else{ //item cannot be picked up
            msg = "I don't think you can pick that up.\n";
        }
    }

    //checking if the player is in a dialogue
    if(getPlayer().isActorInDialogue() && takeSuccess){
        //adding 1 point of anger for every time they do this action
        getActiveDialogueNPC().calculateAnger(BigDecimal.ONE);
        msg += getActiveDialogueNPC().getTakeResponse() + "\n" ;
    }

    return msg;
}
```

The **dropItem** method will check if the named item is found in the player's inventory and if it's able to be dropped (not a key item). If it passes both checks, the item will be transferred from the player's inventory and into the Room's inventory. Note that taking and dropping items while in a dialogue interaction can affect the active dialogue NPC's attributes.

```java
//method to drop an item from the player's inventory
public String dropItem(String itemName){
    String msg = "";
    boolean dropSuccess = false; //checks if the drop action was successful
    Thing t = HelpMethods.thisThing(itemName, getPlayer().getThings());
    if (itemName.equals(anObject: "")){//checks if the itemName is an empty string
        msg = "You have to *name* the object. I can't read your mind.";
    }else if(t == null){//checks if the itemName exists
        msg = "There's nothing named " + itemName + " in your inventory.";
    }else{
        if(t.isThingKey() == false){ //checks if the item is a key item (not droppable)
            transferItem(t, player.getThings(), player.getRoom().getThings());
            msg =  "\n" + HelpMethods.capitalizeString(itemName) + " has been dropped.";
            dropSuccess = true;
        }else{ //key item, item cannot be dropped
            msg = "I don't think you should drop that.";
        }
    }
    //checking if the player is in a dialogue and if the drop was a success
    if(getPlayer().isActorInDialogue() && dropSuccess){

        //add 1 point of anger if the dropped item doesn't originate from the scene
        if(t.getLocation() == getActiveRoom().getLocation()){
            getActiveDialogueNPC().calculateAnger(BigDecimal.ONE.negate());
        }
        //negate 1 point of anger if the dropped item originates from the scene
        else{
            getActiveDialogueNPC().calculateAnger(BigDecimal.ONE);
        }
        msg += getActiveDialogueNPC().getDropResponse() + "\n" ;
    }

    return msg;
}
```

8. Movement

This section will focus on how to program processes the “move” command.

The method **moveActorTo** is the core of how actors are moved around the map, setting their Room attributes when a move is successful.

```java
//method to call the moveTo method for an Actor (NPC, PC)
public void moveActorTo(Actor anActor, Room aRoom) {
    anActor.setRoom(aRoom);
}

//method to call the moveTo method for the player character (PC)
public int movePlayerTo(Direction dir){
    return moveTo(player, dir);

}
```

The **movePlayerTo** method is one specifically called for moving the player character. Alongside the **goN, goS, goW, goE, moveTo, and moveHandler,** they facilitate player movement by checking if a room has the intended directional exit and notifying the player on whether movement was successful or not.

```
//MOVEMENT-------------------------------------------------------------------------
//Methods for moving the player around the map
private void goN() {
    moveHandler(movePlayerTo(Direction.NORTH));
}

private void goS() {
    moveHandler(movePlayerTo(Direction.SOUTH));
}

private void goW() {
    moveHandler(movePlayerTo(Direction.WEST));
}

private void goE() {
    moveHandler(movePlayerTo(Direction.EAST));
}

//method to move an Actor (NPC, PC) to a room
public int moveTo(Actor anActor, Direction aDirection){
    Room activeRoom = anActor.getRoom();
    int exit;
    switch(aDirection){
        case NORTH:
            exit = activeRoom.getNorth();
            break;
        case SOUTH:
            exit = activeRoom.getSouth();
            break;
        case EAST:
            exit = activeRoom.getEast();
            break;
        case WEST:
            exit = activeRoom.getWest();
            break;
        default:
            exit = Direction.NOEXIT;
            break;
    }
    if (exit != Direction.NOEXIT && !anActor.isActorInDialogue()){
        moveActorTo(anActor, map.get(exit));
    }
    return exit;
}
```

```java
//method to try a room/location change
//returns a special message if unsuccessful (NOEXIT)
//returns the scene name + description if successful
private void moveHandler(int roomNumber) {
    String msg;

    //if the character is in a dialogue
    if(getPlayer().isActorInDialogue() && isSpecialActionTriggered == 0){
        getActiveDialogueNPC().calculateAnger(BigDecimal.ONE);
        msg = "You're currently talking to someone, finish up first.\n\n";
        msg += getActiveDialogueNPC().getMoveResponse() + "\n" ;

    }
    //if no exits were found
    else if (roomNumber == Direction.NOEXIT) {
        msg = "You can't travel that way, sorry.\n";
    }else{
        //move is a success, exit was found
        getPlayer().setLocation(roomNumber);
        getPlayer().addVisit(getActiveRoom()); //increments visit value in the hashmap by 1
        msg = "You are now in " + getActiveRoom().getName() + ".\n " + '\n'+ getActiveRoom().getDescription();
    }
    System.out.print(msg);
}
```

9. Setters and Getters

This section will be about the extra setter and getter methods implemented in the class.

There is only one setter method I implemented, that being the **setRoomVisitMap**, to assign the roomVisits HashMap with its keys (Rooms) and values (visit number).

The getter methods are used to return the player character, active dialogue NPCs, the currently active room (the room the player is in), a dialogue option, and names of all the rooms and NPCs.

```java
//SETTERS AND GETTERS----------------------------------------------------------------
//method to return the player object
public Actor getPlayer() {
    return player;
}

//method to return the active room
public Room getActiveRoom(){
    return getPlayer().getRoom();
}

//method for getting the NPC that the player character is interacting with
public Actor getActiveDialogueNPC(){
    return getActiveRoom().getNPC(getPlayer().getWhoActorInDialogueWith());
}

//method to return each of the room's names
public ArrayList<String> getAllRoomNames(){
    ArrayList<String> allRoomNames = new ArrayList<>();
    for(Room s : this.map){
        allRoomNames.add(s.getName().toLowerCase());
    }
    return allRoomNames;
}

//method to return each of the NPC names
public ArrayList<String> getAllNPCNames(){
    ArrayList<String> allNPCNames = new ArrayList<>();
    for(Room s : this.map){
        allNPCNames.addAll(s.getNPCNames());
    }
    return allNPCNames;
}

//method to return one specific dialogue that is an option for the player
public Dialogue getADialogueOption(int index){
    return this.listOfChoices.get(index);
}

//method to assign the Rooms a visit value in the hashmap
public void setRoomVisitMap(){
    for(Room s : this.map){
        playerVisits.put(s, value: 0);
    }
}


//SETTERS AND GETTERS END------------------------------------------------------------
```

## 10. Processing Input

This final section will explain how the program takes the parsed user input and runs the intended commands.

The **runCommand** method will start by determining whether the input has either 1 or 2 words, if not, it will return a message telling the player the command is unrecognized.

```
//method to call the processVerbNoun method and run the player's intended command
public String runCommand(List<String> inputstr) {
    String s;
    if(inputstr.size() == 2){
        s = processVerbNoun(inputstr);
    }else if(inputstr.size() == 1){
        s = processVerb(inputstr);
    }else{
        s = "That's not a recognized action.\n";
    }

    return s;
}
```

If only one word was found (e.g. “about”, “commands”, “quit”, etc), the **processVerb** method will be the one to handle and run the intended commands.

If a command wasn’t recognized, it will return a message notifying the player as such.

The accepted command words:

- “quit” will exit the game
- Numeric values of 1, 2, or 3 will process the attribute changes that happened to the NPC the player is talking to, clearing the listOfChoices to set up for the next option prompt
- “commands” will show a list of available commands in game
- “about” will show a short about text talking about Zwielicht in general

```java
//method for processing one word commands
public String processVerb(List<String> wordList){
    String verb;
    String msg = "";
    verb = wordList.get(index: 0);

    //if the command is quit
    if(verb.equals(anObject: "quit")){
        msg += "Thank you for playing the demo!";
    }
    //if the command is a valid number choice (for dialogue choices)
    else if(getPlayer().isActorInDialogue() && TextParser.isNumberChoiceValid(verb)){
        if(verb.equals(anObject: "1") && getPlayer().isActorInDialogue()){
            msg = String.valueOf(this.getNextDialogueList().get(index: 0));
            //changing the NPC according to the selected dialogue option's attribute changes
            processAttributeChange(dialogueOptionIndex: 0);
            listOfChoices.clear(); //resetting the list of dialogue choices
            System.out.println();

        }else if(verb.equals(anObject: "2") && getPlayer().isActorInDialogue()){
            msg = String.valueOf(this.getNextDialogueList().get(index: 1));
            //changing the NPC according to the selected dialogue option's attribute changes
            processAttributeChange(dialogueOptionIndex: 1);
            listOfChoices.clear(); //resetting the list of dialogue choices
            System.out.println();

        }else if(this.getNextDialogueList().size() == 3 && verb.equals(anObject: "3") && getPlayer().isActorInDialogue()){
            msg = String.valueOf(this.getNextDialogueList().get(index: 2));
            //changing the NPC according to the selected dialogue option's attribute changes
            processAttributeChange(dialogueOptionIndex: 2);
            listOfChoices.clear(); //resetting the list of dialogue choices
            System.out.println();
        }
    }
    //if the command was commands (prints the command list)
    else if(verb.equals(anObject: "commands")){
        showCommands();
    }
    //if the command was about (prints the text talking about the game)
    else if(verb.equals(anObject: "about")){
        showAbout();
    }
    //if the command wasn't able to be recognized
    else{
        msg = "That's not a recognized action.\n";
        return msg;
    }
    return msg;
}
```

If two words were found (a verb + a noun), the **processVerbNoun** method will handle and run the intended commands.
If a verb or noun couldn't be recognized, the method will return a message notifying the player of the fact.

The accepted command words:

- "move" + accepted direction will move the player character that direction if possible
- "examine" + accepted noun will return the appropriate description/examination text for the noun
- "take" + accepted noun will transfer the noun to the player's inventory if possible
- "drop" + accepted noun will transfer the noun to the active room's item list if possible

```java
//method for processing two word commands
public String processVerbNoun(List<String> wordlist){
    String verb;
    String noun;
    String msg = "";
    boolean error = false;
    verb = wordlist.get(index: 0);
    noun = wordlist.get(index: 1);

    //if the command isn't found in the commands list
    if(!commands.contains(verb)){
        msg = "You either made a typo, or " + verb +" isn't a recognized action.\n";
        error = true;
    }
    //if the noun doesn't exist
    if(!allItemNames.contains(noun) && !directions.contains(noun) && !miscNouns.contains(noun) && !allNPCNames.contains(noun)){
        msg += "You either made a typo, or " + noun + " isn't a recognized noun.\n";
        error = true;
    }
    //if the previous checks were passed, checks for which action the player wants to do
    if(!error){
        //processing move actions
        if(verb.equals(anObject: "move") && directions.contains(noun)){ //processing move commands
            switch(noun){
                case "north":
                    goN();
                    break;
                case "south":
                    goS();
                    break;
                case "east":
                    goE();
                    break;
                case "west":
                    goW();
                    break;
                default:
                    msg = "You either made a typo, or " + noun + " isn't a recognized direction.\n";
                    break;
            }
        }
```

```java
        }
        //processing examine actions
        else if(verb.equals(anObject: "examine")){ //processing examine commands
            msg = examineItem(noun);
        }
        //processing take actions
        else if(verb.equals(anObject: "take")){ //processing take commands
            msg = takeItem(noun);

        }
        //processing drop actions
        else if(verb.equals(anObject: "drop")){ //processing drop commands
            msg = dropItem(noun);

        }
    }

    return msg;
}
```

k. Zwielicht Class (Main Driver)

The Zwielicht class is the main driver class for the game. It mostly contains methods that handle saving, loading, and deleting files. It has a private default constructor to avoid instantiation.

It has the following attributes:

- **myDateFormatter**, used to format a DateTime object into a more readable format
- **in,** a BufferedReader object to get user input
- **game**, the a AdventureGame object containing the main logic and code that makes up Zwielicht

```
public class Zwielicht {

    //private constructor to avoid instantiation
    private Zwielicht() {}

    //date and time formatter
    private static DateTimeFormatter myDateFormatter = DateTimeFormatter.ofPattern(pattern: "E, MMM dd yyyy HH:mm:ss");

    //creating a new buffered reader for getting input
    private static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

    //declaring the game object
    public static Game game;
```

It has the following methods:

- **checkSaveExists and checkFolderExists**, for checking if a save file or a player folder already exists, returning a boolean value accordingly.

```
//method for checking if a save file exists
private static boolean checkSaveExists(String playerName, String saveName) {
    boolean exists;
    File file = new File(".\\..\\gamesaves\\" + playerName + "\\" + saveName + ".sav");
    if (file.exists()) {
        exists = true;
    }else{
        exists = false;
    }
    return exists;

}
//method for checking if a player folder exists
private static boolean checkFolderExists(String playerName) {
    boolean folderExist;
    File folder = new File(".\\..\\gamesaves\\" + playerName);
    if (folder.exists()) {
        folderExist = true;
    }else{
        folderExist = false;
    }
    return folderExist;

}
```

- **lowerTrim**, for returning an input to lowercase and with trimmed whitespace

```
//method for formatting string to lower case and trimmed
private static String lowerTrim(String word){
    return word.trim().toLowerCase();
}
```

- **getLastTimeFileModified**, for returning a String containing the date and time of when a file was last modified.

```
//method for getting the last time a file was modified
private static String getLastTimeFileModified(String playerName, String saveName){
    //creating a new file object from the pathName
    File file = new File(".\\..\\gamesaves\\" + playerName + "\\" + saveName + ".sav");
    //storing the long variable from the lastModified method
    long lastModified = file.lastModified();
    //formatting the long into a date and time
    LocalDateTime date = LocalDateTime.ofInstant(Instant.ofEpochMilli(lastModified), ZoneId.systemDefault());
    //using myDateFormatter to change it into a more readable output
    String formattedDate = date.format(myDateFormatter);
    return formattedDate;

}
```

- **currentDateTime**, for returning a String containing the current date and time

```
//method for getting the current local date/time
private static String currentDateTime(){
    LocalDateTime dateNow = LocalDateTime.now();
    return dateNow.format(myDateFormatter);

}
```

- **saveGame,** to serialize a AdventureGame object and save it in a .sav file

```
//method for saving the game
private static void saveGame( String playerName, String saveName){
    try{
        //creating a file output stream object from the path
        FileOutputStream fos = new FileOutputStream(".\\..\\gamesaves\\" + playerName + "\\" + saveName + ".sav");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        //converting the game object into a byte stream (serializing)
        oos.writeObject(game);
        oos.flush();
        oos.close();
        //message to notify the player the save was successful
        System.out.println("\nYour progress was succesfully saved.\n");

    }catch (Exception e){
        //message to notify the player the save failed
        System.out.print("Serialization error. Save was unsuccessful.\n" + e.getClass() + ": " + e.getMessage());
    }
}
```

- **loadGame**, to deserialize a .sav file into a AdventureGame object

```
//method for loading a game save
private static void loadGame( String playerName, String saveName){
    try{
        //creating a file input stream object from the path
        FileInputStream fis = new FileInputStream(".\\..\\gamesaves\\" + playerName + "\\" + saveName + ".sav");
        ObjectInputStream ois = new ObjectInputStream(fis);
        //reading the saved object (deserializing)
        game = (AdventureGame) ois.readObject();
        ois.close();
        //message to notify the player the load was successful
        System.out.println("\nGame successfully loaded.\n");

    }catch (Exception e){
        //message to notify the player thata the load failed
        System.out.print("Serialization error. Load was unsuccessful.\n" + e.getClass() + ": " + e.getMessage());
    }
}
```

- **deleteSave,** to delete a save file

```
//DELETING A SAVE FILE
//method for deleting a save file
private static void deleteSave(String playerName, String saveName) {
    String path = ".\\..\\gamesaves\\" + playerName + "\\" + saveName + ".sav";
    File save = new File(path) ;
    //checks whether a save file was successfully deleted or not
    if(save.delete()){
        System.out.println("Save successfully deleted.");
    }else{
        System.out.println("Save could not be deleted.");
    }
}
```

- **processSave, processLoad, and processDelete**, each are used to process their respective actions.

The differences in the methods lie in the validation of the player folder and printed message content. The **processLoad and processDelete** methods check it in this method, while the **processSave** method checks by calling the **ifSaveExists** method. The printed messages all correspond to which process is being handled (save, load, delete).
Since the process for all three are similar, I'll just explain the **processSave** method :

- Prompts player for player folder name
- Prompts the player for confirmation on the player folder name
- Checks if the player response is valid
    - If it's valid, prompts the player for a save file name
        - If the response is "yes", calls **checkSaveExists** method
        - If the response is "no" or anything else, cancels the process
    - If it's invalid, prints a message notifying the player

```java
//method ffor processing and confirming a save
public static boolean processSave() throws IOException{
    boolean checkFile = false;
    String playerName;
    String saveName;
    List<String> response;

    //prompting the player for the player folder name
    System.out.print("Enter your player name: \n" + "> " );
    playerName = in.readLine();
    //prompting the player for confirmation
    System.out.println(x: "\nWould you like to save this character?");
    System.out.println("Player Name: " + playerName);
    System.out.println(currentDateTime());
    System.out.print(s: "> ");
    response = TextParser.parseInput(in.readLine());

    //checks if a valid response is given
    if(!response.isEmpty()){
        if(response.get(index: 0).equals(anObject: "yes")){
            //response is yes
            //prompting for save file name
            System.out.println(x: "\nEnter the name of your save:");
            System.out.print(s: "> ");
            saveName = lowerTrim(in.readLine());
            playerName = lowerTrim(playerName);
            //checking if a save already exists
            checkFile = checkSaveExists(playerName, saveName);
            ifSaveExists(playerName, saveName, checkFile);
        }else{
            //response is no
            System.out.println(x: "Got it, save process cancelled.");
        }
    }else{
        //unrecognized option
        System.out.println(x: "I can't understand that, save process has been cancelled.");
    }
    return checkFile;

}
```

- **ifSaveExists, ifLoadExists, ifDeleteExists,** each are used to check if a save file exists or not, then call the associated methods (**saveGame, loadGame, and deleteSave**).

Similar to the process methods, the if-blank-Exists methods are quite similar, so I'll only be explaining the **ifSaveExists** method for brevity.

The steps:

- Checks if the save file already exists
    - If it doesn't, creates a new player folder and calls the **saveGame** method
    - If it does, prompts the player if they want to overwrite the save file:
        - If the response is "yes", proceeds to call the **saveGame** method
        - If the response is "no" or invalid, cancels the save process

```
//method for processing if similar save data already exists
public static void ifSaveExists(String playerName, String saveName, boolean checkFile) throws IOException{
    List<String> response;
    if(checkFile){
        //prompting the player wants to overwrite an existing save
        System.out.println("\nA save file with the same name already exists. Overwrite it?");
        System.out.println("Player Name\t: " + playerName);
        System.out.println("Last Save Date\t: " + getLastTimeFileModified(playerName, saveName));
        System.out.print("> ");
        response = TextParser.parseInput(in.readLine());

        //checks if a valid response was given
        if(!response.isEmpty()){
            //save overwritten
            if(response.get(0).equals("yes")){
                saveGame(playerName, saveName);
            }
            //save not overwritten
            else{
                System.out.println("\nGot it, save process cancelled");
            }
        }else{
            //unrecognized response
            System.out.println("I can't understand that, save process has been cancelled.");
        }

    }else{
        //creating a new player folder
        File file = new File(".\\..\\gamesaves\\"+ playerName);
        file.mkdir();
        saveGame(playerName, saveName);
    }
}
```

- **showIntro**, to show the intro to the game

```
//printing out the story intro
public static void showIntro(){
    ReadFile.printFile(".\\..\\gamefiles\\Intro.txt");
}
```

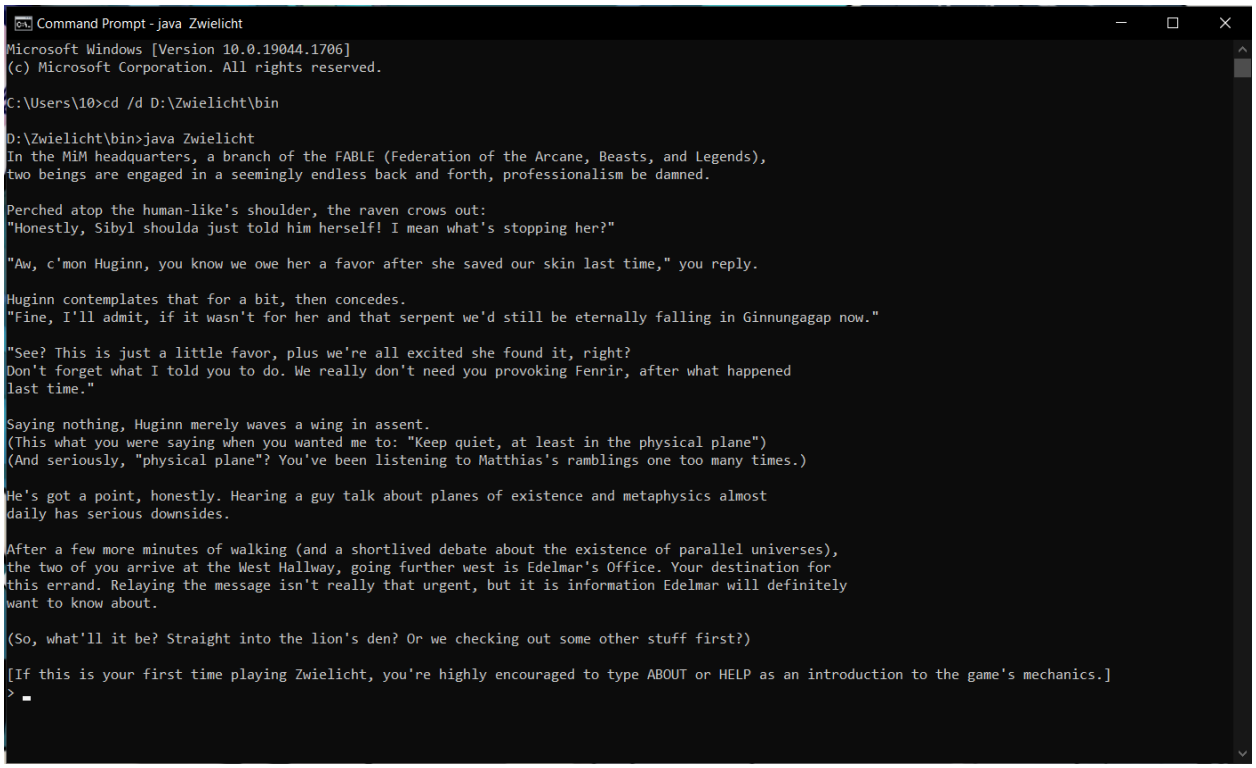
- **Main Driver**

This section covers the main method of Zwielicht. It first shows the intro to the game and creates a new AdventureGame object. After that, comes a do-while loop that keeps running until the command “quit” is given. It also starts the office dialogue scene (the only dialogue interaction in the game right now) when the requirements are met.

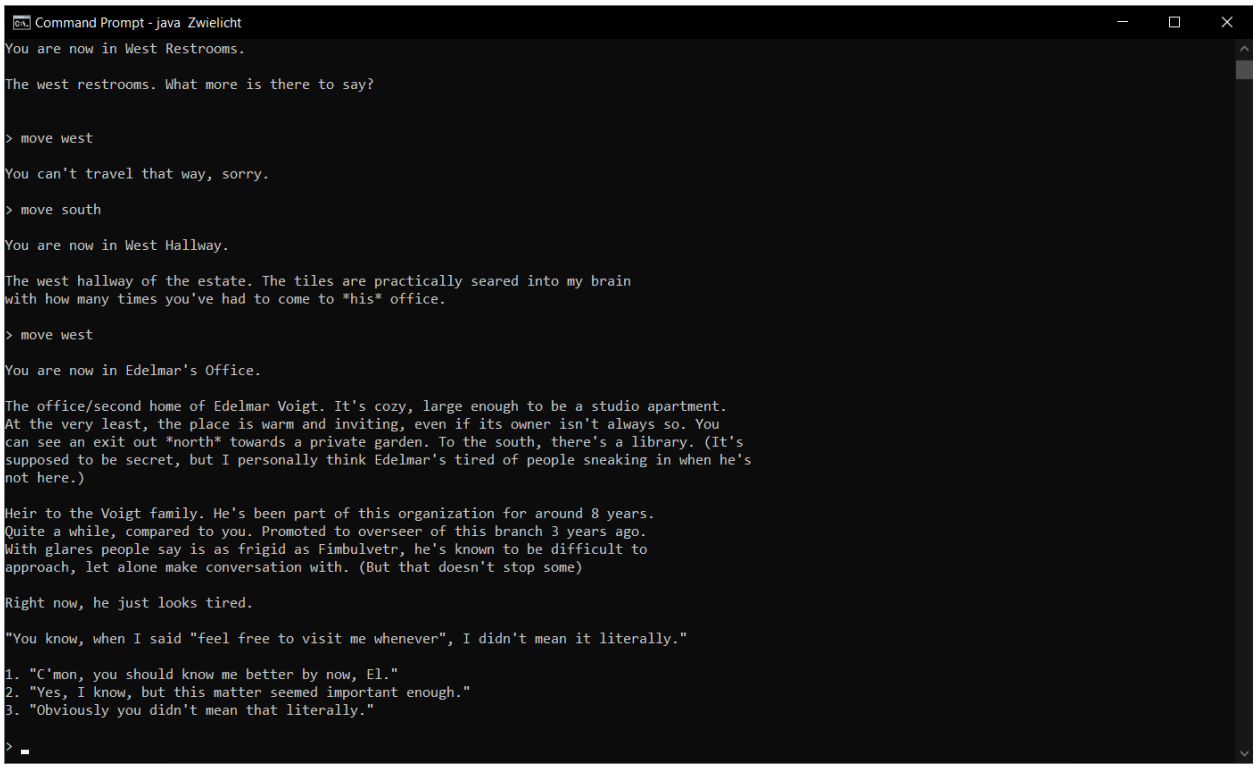
```
// Main driver method
Run | Debug
public static void main(String[] args) throws IOException
{
    showIntro();
    game = new AdventureGame();
    String input;
    String output;
    String checkQuit = "";
    List<String> outputs;

    do {
        output = "";
        System.out.print(s: "> ");
        input = in.readLine();
        outputs = TextParser.processInput(input);
        System.out.println();

        //if the output is empty, returns
        if(outputs.isEmpty()){
            output = "I can't recognize that, sorry.\n";

        }else{
            //checking which command is trying to be run
            checkQuit = outputs.get(index: 0);
            switch (outputs.get(index: 0)) {
                case "save":
                    processSave();
                    break;
                case "load":
                    processLoad();
                    break;
                case "delete":
                    processDelete();
                    break;
                default:
                    output = game.runCommand(outputs);
                    break;
            //playing out the office scene
            }if(game.getPlayer().getLocation() == 1
                && game.getPlayer().getSpecificRoomVisits(game.getPlayer().getRoom()) == 1
                && !game.getPlayer().getRoom().getIfDialogueIsFinished()){
                //starts the dialogue
                game.startDialogue(actorName: "edelmar", fileName: ".\\..\\gamefiles\\Scene.txt");
            }

        }
        //displaying the output
        System.out.println(output);

    } while (!"quit".equals(checkQuit));
}
```

Evidence of Working Program

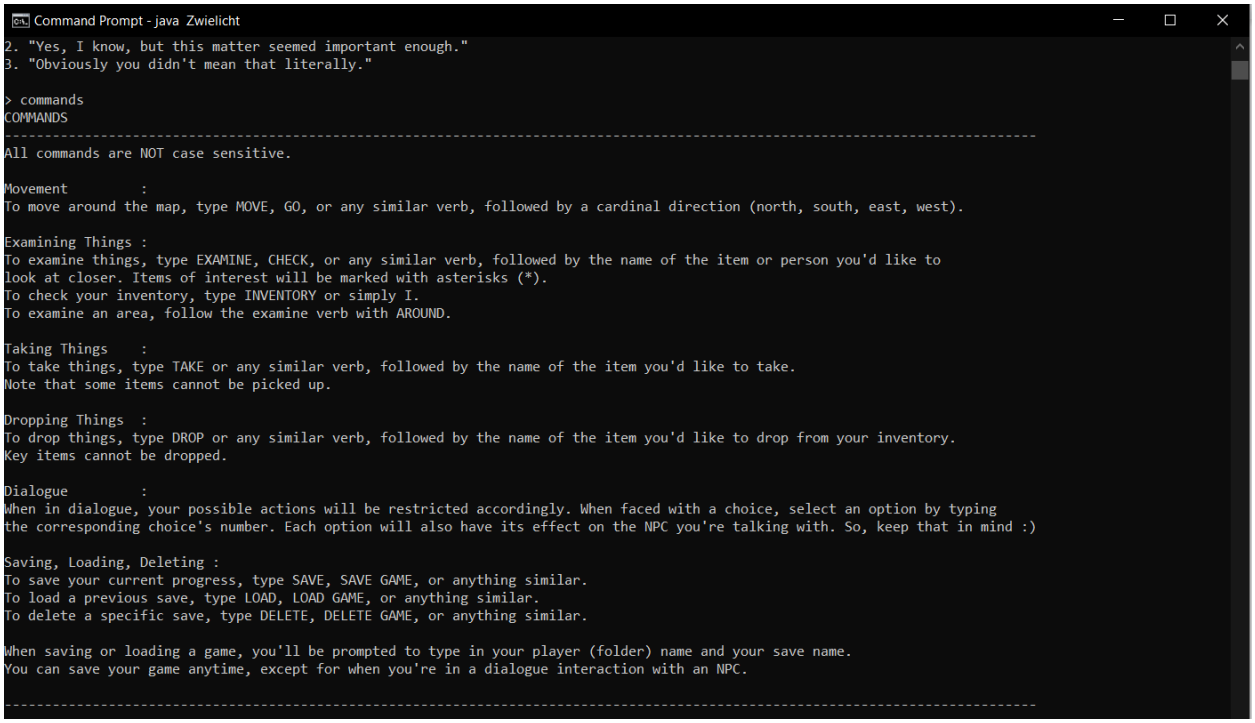
- Intro Scene



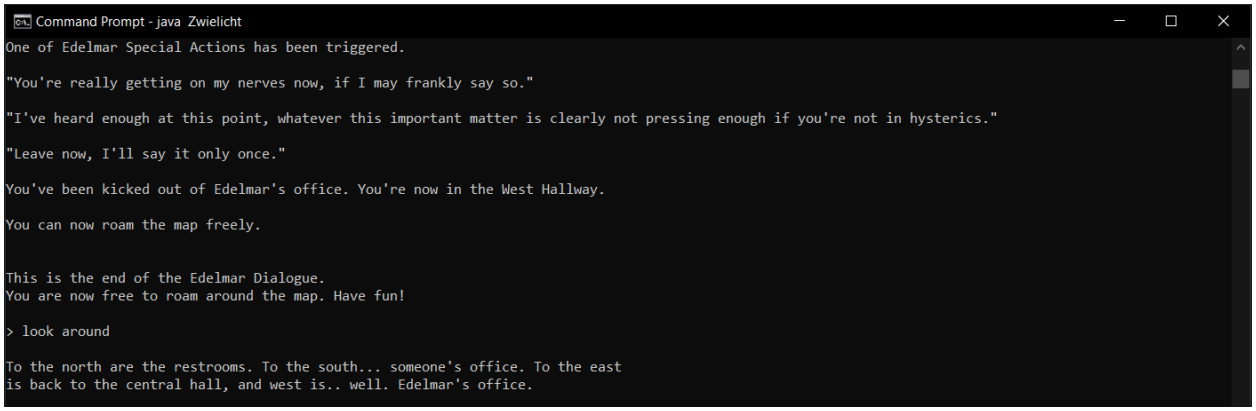

- Moving and engaging in a dialogue interaction



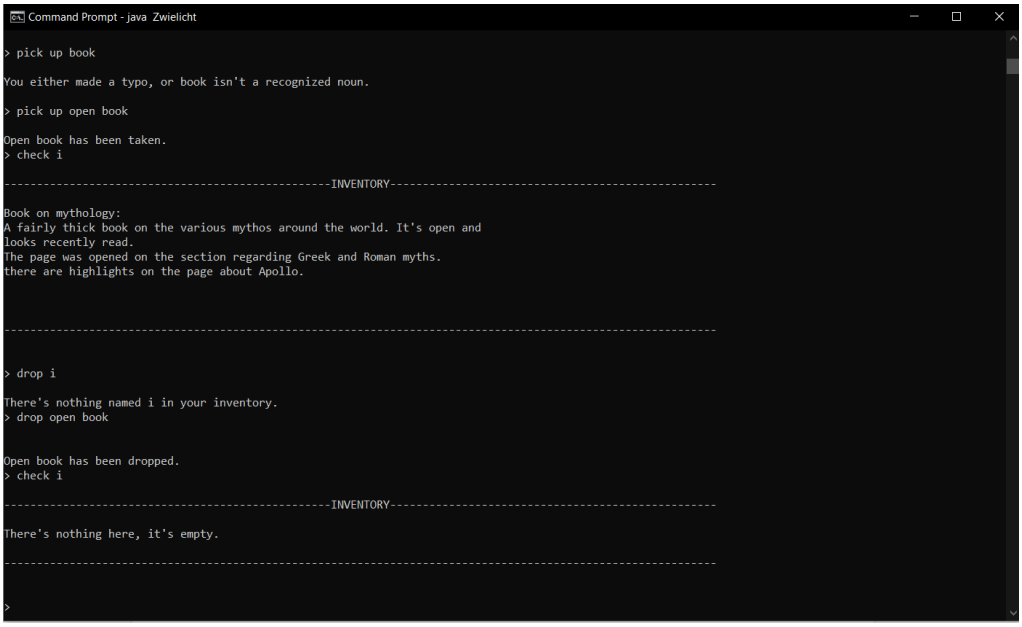
- Commands screen



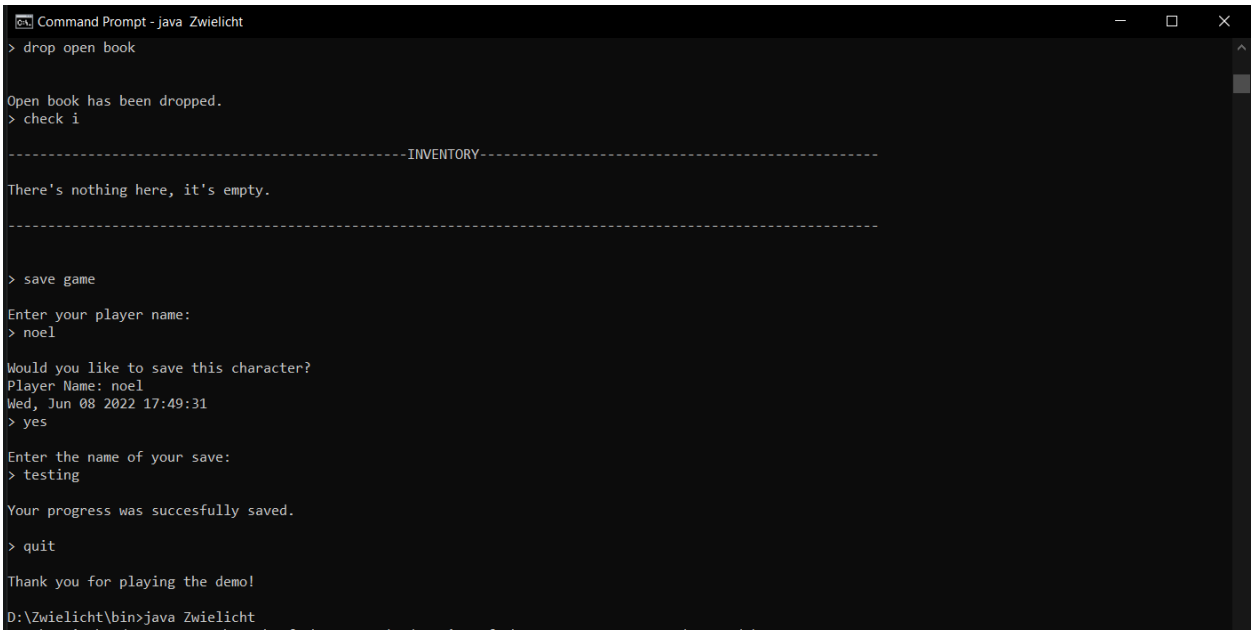

- NPC Special Action

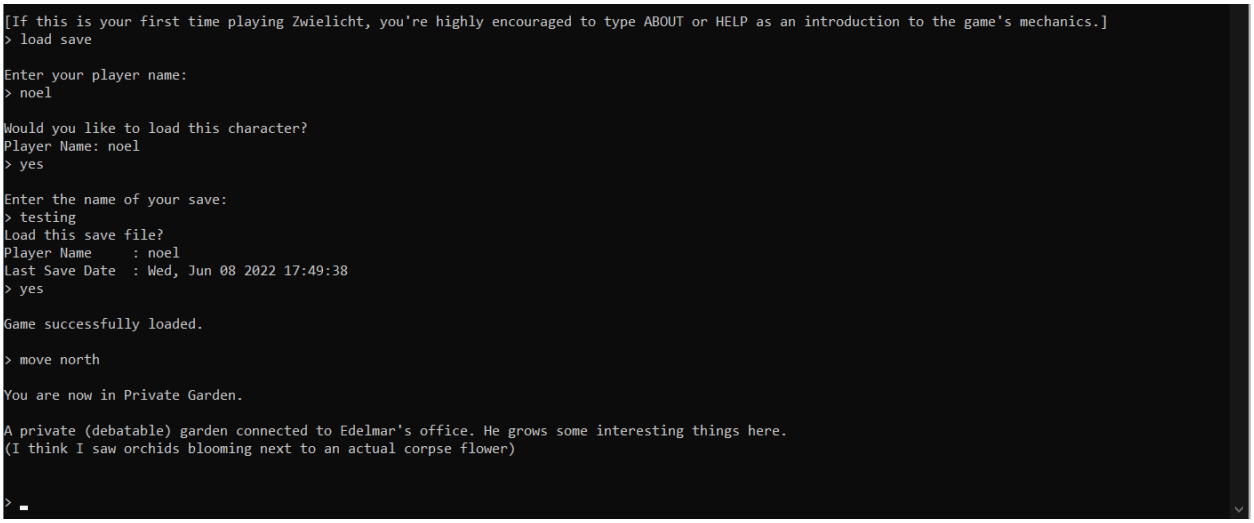



- Taking and Dropping Items



- Saving and loading a game




- Video Link

https://drive.google.com/file/d/1fd0oz3KBlC6wVpG1Djb43RWDk9XivH6e/view?usp=sharing

# Resources

## Links

1. https://emshort.blog/2018/09/11/mailbag-deep-conversation/
2. https://ifdb.org/viewgame?id=a746d3agtfizlx0x
3. https://forum.unity.com/threads/help-coding-dialog-system.577909/
4. https://youtube.com/playlist?list=PLZHx5heVfgEvT5BD8TgLmGrr-V64pX7MD
5. https://github.com/MyreMylar/christmas_adventure
6. https://github.com/CRMinges/adventure
7. https://trizbort.io/

## Technology Used

1. Visual Studio Code, for editing and debugging the code
2. PlantUML, for creating the UML Class Diagram
3. Trizbort.io, for creating the barebones map structure