Katelynn Prater
Professor Kanemoto
CPSC 39
18 July 2025

Programming Project Report: Escape Room

For the final project for this class, I have decided to create an escape room game, complete with three different puzzles that a player must solve, items, proper input handling/parsing, a droll, bad-mannered narrator, and a global state tracker to allow for the creation of many more rooms. (I plan to!) In the game, you begin in the middle of the room, where you begin with dialogue from the narrator, the ability to ask questions, and then a prompt to select a corner in the terminal. Each corner (four) contributes to either a puzzle in its entirety or as a part in another puzzle. Puzzle 1 involves pressing buttons in a specific random order, with a hint given to you in the fourth corner once you press the third button once. Puzzle 2 involves memorization, where you "hear" sounds and must input them back in order following the blanking of the terminal, or the sound order will reroll. Finally, puzzle 3 is simple–you stare at the mirror, discomfited since the player is not quite human–until the mirror shatters. Once all three puzzles are done, the "ending" method is automatically triggered, and you "enter" the second room, where the program ends. This program isn't quite useful, I don't think, unless its use is "entertainment", of which I think it has in spades. As an idea it's unique and contains many facets as stated prior.

For the three algorithms I chose, I chose:

- The input parsing algorithm to get the choice of item to take out and hold from inventory. This algorithm prevents improper input (i.e. try/catch and proper integer checking), changes the heldItem variable (what player is currently holding) with proper input, and then returns the item that the player chose.
- Another input parsing algorithm to determine whether the player has asked a repeat question to the narrator. This iterates over all keys of a LinkedHashMap and compares them to an array ranging from 1->5, returning false when it finds the same input if the LinkedHashMap contains that same key. If a question has been asked, the question is removed from the hashmap.
- An algorithm to output the typewriter style dialogue from the narrator, given a String. It turns the str into an array of characters, and then uses the wait function and a random number generator between 35 and 50 ms to print the next character.

All three algorithms were developed by me, although some assistance was taken from ChatGPT in the form of enlightening me to the wait() command (Algorithm 3) and a Runnable (Algorithm 2). Otherwise, everything else was by me. Regarding the big O notation of each algorithm, they are as follows:

- Algorithm 1: O(k), where k is the number of iterations as decided by the player. It can theoretically go on forever if you input wrongly (while loop), but in practice it is O(1) given there's only if, else, try, and catch statements. Also, in getInventory(), ArrayLists have O(1) transversal time.

- Algorithm 2: O(1), as the time of the algorithm doesn't depend at all of the size of the LinkedHashMap. The for loop can make it seem this is the case, at least in the worst case, but it is *always* iterating over a fixed array of size 5. It also doesn't matter, but the .containsKey() method is also typically O(1).
- Algorithm 3: O(n), as the time of this algorithm merely depends on the size of the String you pass. This algorithm loops over all characters, and outputs them in randomly decided millisecond pauses. While this does noticeably impact how quick the algorithm terminates, it doesn't meaningfully change the O, even if it was, say, fifty minutes. It still increases in proportion to the size of the String.

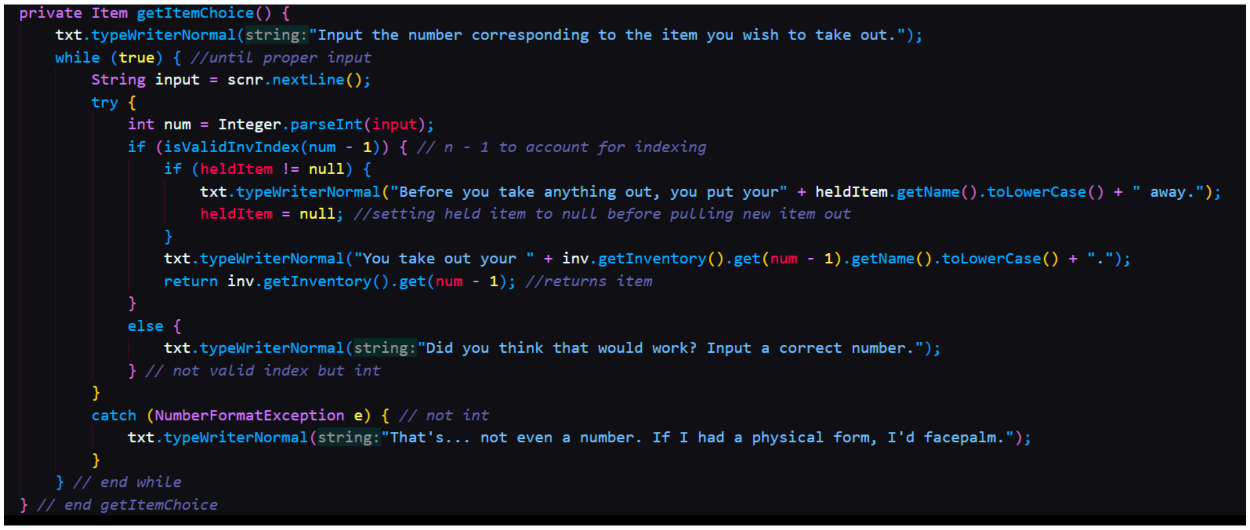
In terms of the data structures used in this project, I used many. Most frequently, I used arrays and ArrayLists, but I also used LinkedHashMaps (not normal HashMaps, as I needed order guaranteed) and I created a custom data structure implementation of a pair of numbers. I used normal arrays in many places, most notably in RoomState where I stored the "started" and "finished" booleans in a Boolean[3] array, as every room will have exactly three puzzles. I use normal arrays when its size is something I know to be fixed beforehand, and RoomState was an excellent usage of it. Next, I use ArrayLists when I want to create a list with variable size. For this, one of its usages was in my buttons puzzle, where I used an ArrayList to store all of the player's inputs (1 for button 1, 2 for button 2, etc.) to compare to another ArrayList that stores the solution. (Since the solution would always have 3 values, I could've easily used a normal array, and that was my immediate impulse, but it made comparing the arrays more cumbersome.) Finally, I used LinkedHashMaps when I want to store key, value pairs (I became very familiar with the concept in a project I did in Python with dictionaries) in a manner that preserves order. With the HashMap, I stored in the key the integer number associated to the question, and in the value the Pair implementation of a String (the question in "num: Question?" format) and Runnable (typeWriter() print statement for the narrator's answer). Order mattered, because even if I remove a key value pair (which I did if a question was asked and answered), I wanted the prompt to still be in ascending order.

In terms of where I found an opportunity, it was in the creation of the StateTracker class. Previously, I had implemented the tracking of all puzzle startings, finishings, and other state variables individually within each class. I noticed the redundancies, and it also became quite cumbersome to work with due to some circular issues. I had previously stored gameStates in the rooms themselves, and I needed the info from the GameState initialized in the room put into the Paper object, so I would pass it the Room, but I also needed to initialize the Paper object in the room itself. So, I decided to create a global StateTracker to fix this. This, also, could probably count as my error, as I spent some unfortunate time trying to work with it, but it got too problematic for me. But, if I need another issue I came across and fixed, it was an issue where it seemed like the loop for the ButtonPuzzle wouldn't terminate. I spent a fruitless 30 or so minutes trying to figure it out, pasting the code into ChatGPT, and every "suggestion" it gave didn't work. It also seemed at a loss. However, it turned out, I was too laser-focused on the method itself and should've remembered that I had put two instantiations of the ButtonPuzzle in

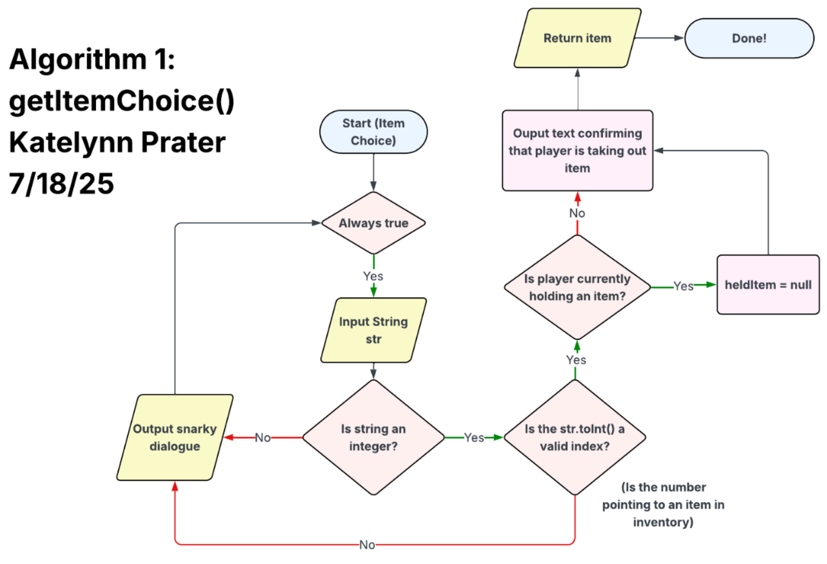
Main to test two different iteration states. I just forgot. It *was* terminating, It just immediately started a brand new one right after. I saw it, face palmed, removed that extra command, and in that specific instance it wasn't my method that was buggy. It was my memory!

In terms to what I would change or add to my game, the natural extension is already pretty much planned. I'm planning on adding four more rooms, as well as fleshing out the iteration mechanic where you go through these rooms over and over with a narrator who's as stuck with you as you are with the facility. I am already planning to do this, and the Flashlight class is evidence of this as it is for room three!

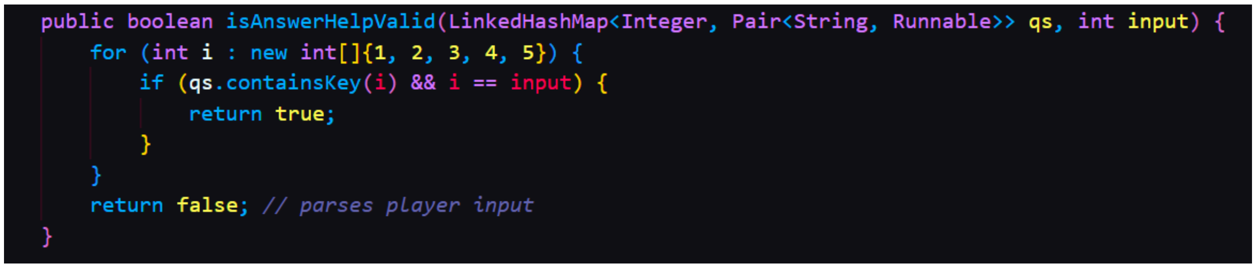
Here are the algorithm code snippets and the respective flowcharts:



```java
private Item getItemChoice() {
    txt.typeWriterNormal(string:"Input the number corresponding to the item you wish to take out.");
    while (true) { //until proper input
        String input = scnr.nextLine();
        try {
            int num = Integer.parseInt(input);
            if (isValidInvIndex(num - 1)) { // n - 1 to account for indexing
                if (heldItem != null) {
                    txt.typeWriterNormal("Before you take anything out, you put your" + heldItem.getName().toLowerCase() + " away.");
                    heldItem = null; //setting held item to null before pulling new item out
                }
                txt.typeWriterNormal("You take out your " + inv.getInventory().get(num - 1).getName().toLowerCase() + ".");
                return inv.getInventory().get(num - 1); //returns item
            }
            else {
                txt.typeWriterNormal(string:"Did you think that would work? Input a correct number.");
            } // not valid index but int
        }
        catch (NumberFormatException e) { // not int
            txt.typeWriterNormal(string:"That's... not even a number. If I had a physical form, I'd facepalm.");
        }
    } // end while
} // end getItemChoice
```



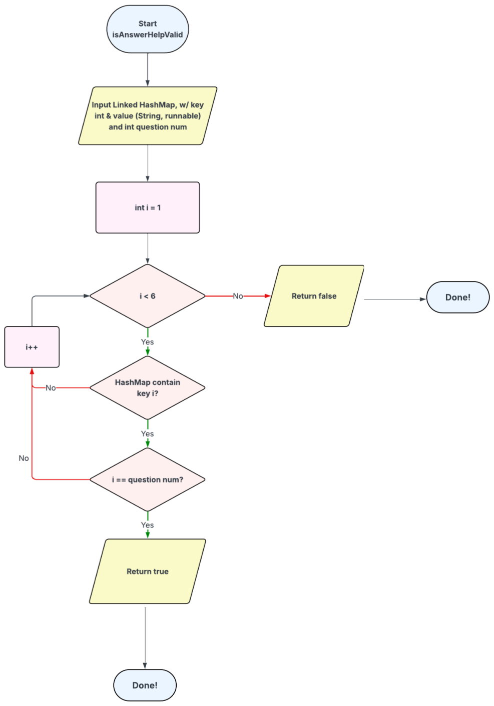
Algorithm 1



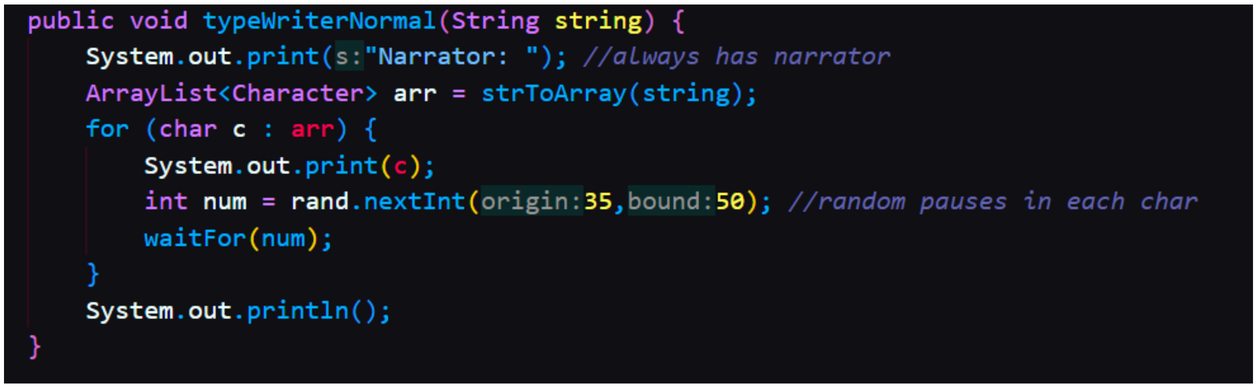
```java
public boolean isAnswerHelpValid(LinkedHashMap<Integer, Pair<String, Runnable>> qs, int input) {
    for (int i : new int[]{1, 2, 3, 4, 5}) {
        if (qs.containsKey(i) && i == input) {
            return true;
        }
    }
    return false; // parses player input
}
```

Final Project
Algorithm 2

Katelynn Prater
7/18/25



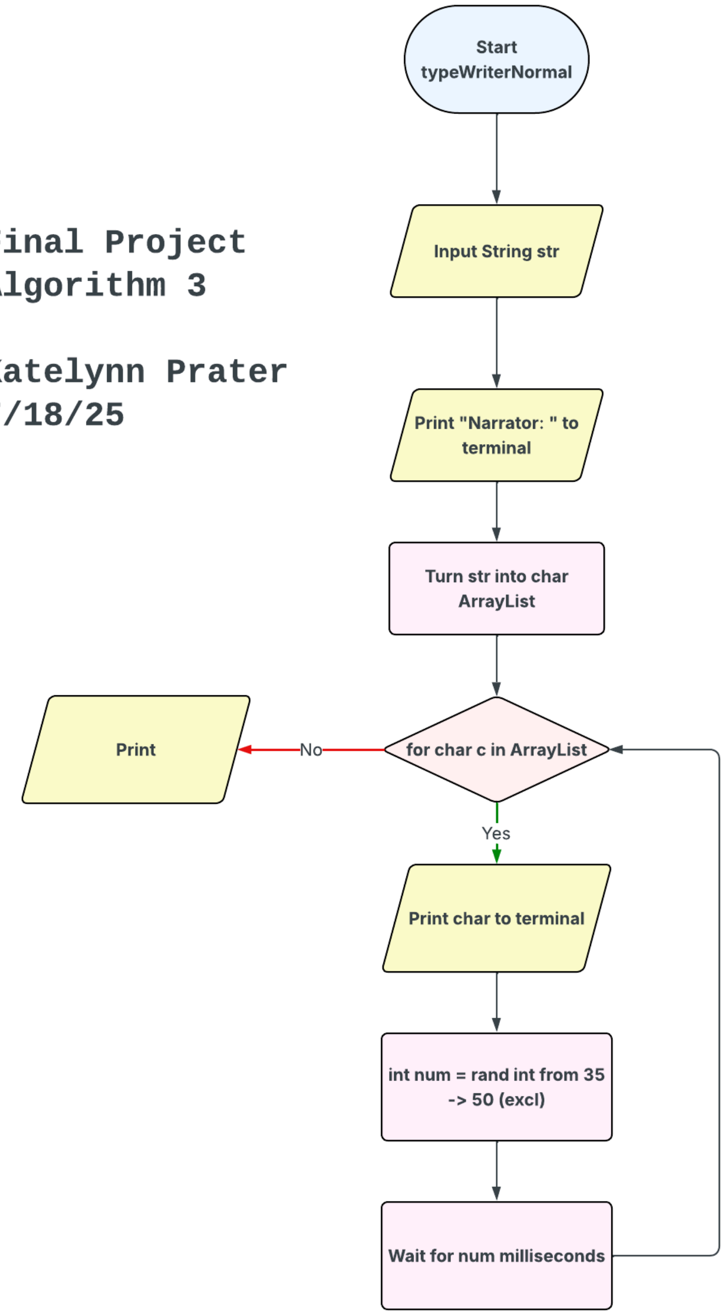
Algorithm 2



```java
public void typeWriterNormal(String string) {
    System.out.print(s:"Narrator: "); //always has narrator
    ArrayList<Character> arr = strToArray(string);
    for (char c : arr) {
        System.out.print(c);
        int num = rand.nextInt(origin:35,bound:50); //random pauses in each char
        waitFor(num);
    }
    System.out.println();
}
```

Final Project
Algorithm 3

Katelynn Prater
7/18/25


Start typeWriterNormal
Input String str
Print "Narrator: " to terminal
Turn str into char ArrayList
for char c in ArrayList
No
Print
Yes
Print char to terminal
int num = rand int from 35 -> 50 (excl)
Wait for num milliseconds

Algorithm 3