Garret Clark

CPSC-39-12801

7 – 18 – 25

Final Project Report

What does your game do, and how's it useful or entertaining?

For my final project, I created a simple game where you can roll for items of varying rarities, and add them into your inventory to view. I wanted to create a game like this for my final project, as I'm planning on creating some simple games with friends in the future, and an inventory system similar to this would be useful to get experience programming for.

Includes at least 3 algorithms as steps or a flowchart, and a snapshot of the algorithms code.

The first algorithm I'll mention is the user input loop in the main class. When the program is initally run, it follows these steps:

1. Displays "Welcome to the inventory game"
2. Displays "To roll for a new item, type 'R'. To display inventory, type 'I'. To show recent pickups, type 'H'. To quit, type 'Q'."
3. Takes in user input. Depending on the input, the algorithm then either rolls for a new item using the RandomItemChooser method, displays the inventory using the InventoryDisplay method, shows recent pickups using the displayRecentPickups method, or quits the program.
4. After the action is completed (unless you quit), the program loops back to take in another user input for an action above.
5. If the program quits, the program displays "Thanks for playing!

Algorithm 1 Code Snapshot:

```java
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        System.out.println("Welcome to the inventory game.\n");

        Scanner scnr = new Scanner(System.in);
        String userInput = "";

        // While Q isn't pressed to quit
        while (!userInput.equalsIgnoreCase("Q")) {
            userInput = DefaultMessage(scnr);

            if (userInput.equalsIgnoreCase("R")) {
                System.out.println("\nRolling for a new item...");
                String newItem = RandomItemChooser.getItemByRarity(
                    RandomItemList.getCommonItems(),
                    RandomItemList.getUncommonItems(),
                    RandomItemList.getRareItems()
                );

                // Add item to inventory and recent pickups
                Inventory.addItem(newItem, scnr);
                PickupTracker.addPickup(newItem);
            }
            else if (userInput.equalsIgnoreCase("I")) {
                InventoryActions.InventoryDisplay(Inventory.inventory);
            }
            else if (userInput.equalsIgnoreCase("H")) {
                PickupTracker.displayRecentPickups();
            }
            else if (!userInput.equalsIgnoreCase("Q")) {
                System.out.println("Unknown command.");
            }
        }

        System.out.println("Thanks for playing!");
        scnr.close();
    }

    public static String DefaultMessage(Scanner scnr) {
        System.out.println("\nTo roll for a new item, type 'R'. To display inventory, "
            + "type 'I'. To show recent pickups, type 'H'. To quit, type 'Q'.");
        return scnr.next();
    }
}
```

The second algorithm I'll mention is the recent items queue in the PickupTracker class, which follows these steps:

1. The history of items is only tracked up to the five most recent.
2. A Queue data structure of type String is declared named recentItems, creating a new LinkedList.
3. The addPickup method: If the queue already has more than five items, remove the oldest item in the queue with the newest item.
4. When the user inputs to display their recent pickups, the displayRecentPickups method will output the queue of their five most recent items, displaying as "Recent pickups: " [itempickups]. If no items have been picked up yet, the program will display: "No recent pickups."

Algorithm 2 Code Snapshot:

```java
import java.util.LinkedList;
import java.util.Queue;

public class PickupTracker {
        // Only tracks the 5 recent items
    private static final int MAX_HISTORY = 5;
    // Declaring a queue and linkedlist
    private static Queue<String> recentItems = new LinkedList<>();

    // Add an item to the recent pickups queue
    public static void addPickup(String item) {
        if (recentItems.size() >= MAX_HISTORY) {
            recentItems.poll();  // Remove oldest item
        }
        recentItems.offer(item);
    }

    // Display recent pickups
    public static void displayRecentPickups() {
        if (recentItems.isEmpty()) {
            System.out.println("No recent pickups.");
            return;
        }
        System.out.println("Recent pickups:");
        int i = 1;
        for (String item : recentItems) {
            System.out.println(i + ": " + item);
            i++;
        }
    }
}
```

The third algorithm, and most interesting one I'll mention is the random item chooser. This is contained in the RandomItemChooser class, and the algorithm is broken down into these steps:

1. A method is created, called getItemByRarity, using the ArrayLists of commonItems, uncommonItems, and rareItems.
2. When the user inputs to roll, the getRandomItem method will roll for random number between 1-100.
3. If the number is > 30, the item is common. If the number is > 10, the item is uncommon. If the number is < 10, the item is rare. After the rarity is determined, the algorithm will then fetch a random object from the correlating ArrayList of the item's rarity, and display the item's rarity. As an example, if you were to roll a common item, the program would display as: Cobwebs (Common). This item is then added to the user's inventory through Main.

Algorithm 3 Code Snapshot:

```java
import java.util.Random;
import java.util.ArrayList;

public class RandomItemChooser {

    private static final Random random = new Random(); // single Random instance

    // This method selects an item based on rarity chance
    public static String getItemByRarity(ArrayList<String> commonItems, ArrayList<String> uncommonItems,
            ArrayList<String> rareItems) {

        int randomNumber = random.nextInt(100) + 1;  // 1 to 100
        String selectedItem;
        String rarity;

        if (randomNumber > 30) {
                selectedItem = getRandomItem(commonItems);
                rarity = "Common";
        }
        else if (randomNumber > 20) {
                selectedItem = getRandomItem(uncommonItems);
                rarity = "Uncommon";
        }
        else {
                selectedItem = getRandomItem(rareItems);
                rarity = "Rare";
        }

        return selectedItem + " (" + rarity + ")";
    }

    // Helper method to get a random item from a list
    public static String getRandomItem(ArrayList<String> items) {
        return items.get(random.nextInt(items.size())); // use the shared Random
    }
}
```

Discusses the Big O time of these algorithms.

For this, I used ChatGPT to answer, as I'm not familiar with the 'Big O' time of algorithms. I hope that's okay to do – and if not, feel free to dock me points for doing so.

1. For the main algorithm:

`main()` loop and logic

- **User input loop**: `O(n)` where n is the number of user commands before quitting.
  - Each loop iteration includes constant-time decisions (`if`/`else if`) and method calls that are analyzed separately.
- Overall: **Depends on input length**, but each command's complexity is low (`O(1)` to `O(m)` where `m` is inventory size).

2. For the PickupTracker algorithm:

`addPickup(String item)`

- `recentItems.size()` → `O(1)`
- `recentItems.poll()` (remove head) → `O(1)`
- `recentItems.offer(item)` (add to tail) → `O(1)`

Total:

✅ `O(1)` — All operations are constant time in a `LinkedList`-based queue.

3. For the RandomItemChooser algorithm:

`getItemByRarity(...)`

- `random.nextInt(100)` → `O(1)`
- One conditional path executes:
  - One call to `getRandomItem(...)` → `O(1)`

Total:

✅ `O(1)`

An explanation of the data structures that you used, why you chose them, and how they were used.

For this program, I used numerous ArrayLists for the user's inventory, and for the item types (across different rarity categories). I chose to use an ArrayList for these, as they're incredibly useful for storing objects which can be used and configured across different classes and through different methods.

I used the item pool ArrayLists (such as common, uncommon, rare, etc.) to have one be added to the user's inventory ArrayList upon rolling. I couldn't use a typical Array for this, as it doesn't store objects, whereas ArrayList do.

I also used a Queue and LinkedList for the PickupTracker class, which made things more simple and organized. Compared to using an ArrayList for this which would have to sort through every object, a Queue simply tracks these items, taking less memory. I used a LinkedList to help implement the queue as they practically go hand-in-hand, and it tracks the first and last elements of the queue.

Explain a step in the design or development process where you encountered an opportunity, and how you used this.

I wasn't entirely sure how to implement a data structure other than an ArrayList into this game, but I had a great idea to track the user's recent items using the Queue data structure. The Queue is literally named to do this, so implementing it helped me fulfill the requirements of having three data structures!

Explains a step in the design or development process where you encountered an error and how you resolved this.

I had a lot of issues getting methods to work from other classes, as I haven't worked with a program using so many classes before. This was something I struggled with all throughout working on the program, so I used ChatGPT to help me with defining the different methods throughout other classes and to help make sure everything worked with each other.

Explains what you would change or add in the next version of your game or app.

I would want to add a feature to remove items before the inventory limit is reached – unfortunately, I didn't have time to implement that although it should be a basic feature. I'd also want to possibly add a way for users to combine two items into a better one, and for users to compare inventories. That would be a fun thing to do!