Task Title: Item System Development Task

Objective

Develop an Item Upgrade System in Java for a game where items have rarity and upgrade mechanics. This task is designed to evaluate your Java programming skills, OOP principles, data structure usage, and code cleanliness.

Task Overview

In this task, you are required to create an Item Upgrade System that allows the creation, management, and upgrading of items based on their rarity. Items can be upgraded to higher rarities by combining multiple items of the same type, following specific upgrade rules. The system should simulate an inventory where players can manage their items, perform upgrades, and view their collection.

This system is a crucial component of a game, and your implementation will demonstrate how well you can design clean, efficient, and functional code for a real-world scenario.

Task Requirements

1. Item System Overview

You need to create a system that supports the creation, management, and upgrading of game items. Items should have:

- **Name:** A string representing the item's name.
- **Rarity:** One of the following:
  - **Common**
  - **Great**
  - **Rare**
  - **Epic**
  - **Legendary**
- **Upgrade Count:** An integer to track the number of upgrades for `Epic` items (e.g., `Epic 1`, `Epic 2`). This count is reset to 0 after the item transitions to a new rarity **(e.g., `Epic 2` to `Legendary`).**

2. Item Upgrade Mechanics

General Rules

To help you understand how the upgrade system works, here is an example:

- Suppose you have three **Common** *Iron Swords* in your inventory. You can upgrade one of them to a **Great** *Iron Sword* by combining it with the other two **Common** *Iron Swords*.
- Similarly, if you have three **Rare** *Iron Swords*, you can upgrade one of them to an **Epic** *Iron Sword* by combining it with the other two **Rare** *Iron Swords*.

Now, let's dive into the detailed rules:

- To upgrade an item, combine the target item with additional items of the same rarity and type.
- Upgraded items advance to the next rarity tier.

Upgrade Requirements

1. To upgrade a **Common** item to a **Great** item, combine it with 2 additional **Common** items of the same type.
2. To upgrade a **Great** item to a **Rare** item, combine it with 2 additional Great items of the same type.
3. To upgrade a **Rare** item to an **Epic** item, combine it with 2 additional **Rare** items of the same type.
4. To upgrade an **Epic** item to **Epic 1**, combine it with any other **Epic** item.
5. To upgrade an **Epic 1** to **Epic 2**, combine it with any other **Epic** item. This process converts the first **Epic 1** into **Epic 2**.
6. To create a Legendary item, combine an **Epic 2** item with 2 additional **Epic 2** items of the same type.

Notes for Epic and Above

- **Epic** items can be upgraded incrementally: **Epic** → **Epic 1** → **Epic 2**.
- To upgrade to **Epic 1**, use any other **Epic** item as the ingredient.
- To upgrade **Epic 1** to **Epic 2**, use **Epic 1** and any other Epic item. This process converts the first **Epic 1** into **Epic 2**.
- To create a Legendary item, combine 1 **Epic 2** item with 2 same **Epic 2** items. All 3 **Epic 2** items must be of the same type (e.g., same name).

3. Functional Requirements

Item Management

- Implement an inventory system to store items.
- Allow items to be grouped and upgraded within the inventory.

Item Creation

- Provide functionality to create new items with a specified rarity and name.

Upgrade Simulation

- Implement an upgrade feature that allows the combination of items based on the rules above.
- Ensure that items used in upgrades are removed from the inventory.

Inventory Display

- Display the current inventory and items in a clean, formatted manner in the console.
- Group items by their rarity.

Error Handling

- Handle cases where the required items for an upgrade are unavailable.
- Ensure invalid operations (e.g., upgrading without enough items) are gracefully handled with clear error messages.

4. Additional Features (Optional)

- Random item generation with weighted probabilities for rarities (e.g., **Common**: 50%, **Great**: 25%, **Rare**: 15%, **Epic**: 8%, **Legendary**: 2%).
- Save and load inventory from a file.

Deliverables

1. **Code:**
    - Provide the Java source files in a GitHub repository.
2. **README:**
    - Include instructions on how to compile and run your program.
    - Highlight any assumptions or design choices you made.
3. **Execution:**
    - Your program should be executable via the command line or IDE.

Evaluation Criteria

Code Quality

- Use of OOP principles.
- Modular design and separation of concerns.
- Clean, readable, and well-documented code.

Functionality

- Correct implementation of the upgrade mechanics.
- Proper inventory management.
- User-friendly console output.

Efficiency

- Appropriate use of data structures to manage and access items efficiently.

Edge Cases

- How well does your code handle invalid inputs or scenarios?

Time Management

- Your ability to deliver a functional prototype within the given time.
- Late submissions are accepted but will incur a penalty.

Time Limit: 3-4 hours. You can submit your solution after the time limit, but late submissions will incur a penalty.

Good luck, and have fun crafting your way to glory! We can’t wait to see the magic you create. 🎮✨