Textual RPG Game System

Objective:
Construct an object-oriented Textual RPG (Role-Playing Game) system in JavaScript ES6, focusing on encapsulating data and behaviors, demonstrating inheritance, implementing polymorphism, and using getters/setters with private properties.

Exercise 1: The Character Superclass

1.1 Define a `Character` class. The constructor should accept three parameters: `name`, `level`, and `health`.

1.2 Inside the constructor, initialize private instance variables for `name`, `level`, `health`, and `inventory`. `inventory` should be an empty array to hold character items. Use the `#` prefix to denote private properties.

1.3 Implement getters for `name`, `level`, and `health` to allow read access to these properties.

1.4 Implement setters for `name`, `level`, and `health` to allow write access to these properties. If the value of `health` is negative, set it to `0`. Validate the other values (e.g., `level` should not be set to a negative number and the `name` is not an empty sting).

1.5 Add a method named `addItem` which accepts a parameter `item`. This method should validate weather the item is an instance of an `Item` class (see exercise 3), and then add it to the private `inventory` array. If the validation fails, it should log an error.

1.6 Implement a method to calculate the total value of the inventory according to the items values. If the inventory is empty, it should return 'Inventory is empty'. This method should not be directly accessible from outside the class (make it a private method and create a getter to expose the calculated total inventory value).

1.7 Create a public method `displayCharacter` that returns a string containing the character's name, level, health, and total inventory value. Use getters within this method to access private properties.

Exercise 2: The Warrior and Mage Subclasses

2.1 Create a `Warrior` subclass that extends `Character`. Its constructor should accept additional parameters for `strength` and `defense`. Initialize these as private properties.

2.2 Implement getters for the `strength` and `defense` properties.

2.3 Override the `displayCharacter` method to include the warrior's `strength` and `defense` along with the details provided by the superclass. Ensure you're using getters to access private properties.

2.4 Define a `Mage` subclass with additional parameters in its constructor for `intelligence` and `mana`. Initialize these as private properties.

2.5 Provide getters for the `intelligence` and `mana` properties.

2.6 Implement setters for `strength`, `defense` (in `Warrior`), and `intelligence`, `mana` (in `Mage`). Validate the new values to ensure they are not negative.

2.7 Override `displayCharacter` in `Mage` to include the `intelligence` and `mana` values along with the superclass details, using getters for private property access.

Exercise 3: The Inventory Item Classes

3.1 Define an `Item` superclass. Its constructor should accept parameters for `name` and `value`. Initialize these as private properties.

3.2 Implement getters for `name` and `value`.

3.3 Implement a setter for `name`. It should validate whether the `newName` is a `string` and is not an empty string. Implement a setter for the `value` property. It should validate whether the `newValue` is a `number` and is greater than 0. If any validation fails, log an error message.

3.4 Create a `Weapon` subclass that extends `Item`. Add additional parameters for `damage` and `type` (e.g., 'sword', 'axe'). Initialize these as private properties.

3.5 Implement getters for the `damage` and `type`.

3.6 Create an `Armor` subclass that extends `Item`. Add additional parameters for `protection` and `material` (e.g., 'leather', 'steel'). Initialize these as private properties.

3.7 Implement getters for `protection` and `material`.

3.8 Implement setters for `damage`, `type` (in `Weapon`), and `protection`, `material` (in `Armor`). Validate the new values to ensure they are not negative or not empty strings, accordingly.

Exercise 4: The Game Class

4.1 Define a `Game` class. Its constructor should initialize a private property `characters` as an empty array.

4.2 Implement an `addCharacter` method that accepts a `character` object and adds it to the `characters` array. Validate whether the new character is an instance of the `Character` class.

4.3 Initialize a private property `currentCharacter` to manage the active character during the game.

4.4 Implement a setter for `currentCharacter` that allows changing the active character by passing a character name. It should search for a character by `name` in the `characters` array and set it as the `currentCharacter` if found. Log an error if the character is not found.

4.5 Calculate damage method

- Implement a method to calculate the damage based on characters' attributes and classes.
- Use the characters' `level`, `strength` (for Warriors), `intelligence` (for Mages), `defense` (for Warriors), and `mana` (for Mages) in the damage calculation.
- Incorporate a defense factor that reduces the damage based on the defender's attributes.
- Ensure the damage is at least 1 to guarantee that every attack has an effect.

Steps:

1. **Define the Method**: Inside the `Game` class, define a method named `calculateDamage`. This method should accept two parameters: `attacker` and `defender`, both of which are instances of the `Character` class or its subclasses (`Warrior`, `Mage`).

2. **Identify Character Classes**: Determine the class of both the `attacker` and `defender` using `instanceof`. This will help in deciding which attributes to use for calculating the damage.

3. **Calculate Base Damage**:
   - For a `Warrior` attacker, calculate the base damage using the attacker's `strength` times the `level`.
   - For a `Mage` attacker, calculate the base damage using the attacker's `intelligence` times the `level`.

4. **Apply Defense Factor**:
   - For a `Warrior` defender, reduce the damage by a factor based on the defender's `defense`, like this: `1 - (defender.defense / 100)`.
   - For a `Mage` defender, reduce the damage by a factor based on the defender's `mana`. Like this: `1 - (defender.mana / 200)`.
   - Apply this defense factor to the base damage calculated in the previous step.

5. **Finalize Damage Calculation**: Calculate the final damage by multiplying the defense factor with the base damage. Ensure the final damage is rounded appropriately and that it is at least 1 to ensure every attack makes an impact.

6. **Return the Damage**: Return the calculated damage from the method.

4.6 Battle Method

Implementation Steps:

1. **Verify Characters**: Ensure that both the `currentCharacter` and the `opponent` are valid `Character` instances. If not, log an error

message.

2. **Initiate Dice Roll**: Simulate a dice roll for both the `currentCharacter` and the `opponent` to determine the initiative or the chance of hitting. This could be as simple as generating a random number between 1 and 20 for each character, mimicking a D20 roll common in RPGs.

3. **Calculate Damage**: Modify the damage calculation to incorporate the result of the dice roll. The character with the higher roll gets to hit. The damage could still depend on the character's class attributes (e.g., `strength` for warriors, `intelligence` for mages), but you might add or multiply the damage by a factor of the dice roll for added effect.

4. **Apply Damage**: Calculate damage by calling the `calculateDamage` method and pass to it the `attacker` and the `defender`.

5. **Determine Outcome**: The battle outcome is now influenced by the result of the damage calculation. Subtract the damage from the `defender`'s health. If an defender's health drops to 0 or below, the character who inflicted the final blow wins, his level is growing by 1 and the opponent's level is subtracted by 1. Otherwise, the battle can continue with another round of dice rolls until there's a decisive outcome.

6. **Return Result**: Return a message with the battle outcome, detailing the winner, the losing character, and the remaining health of the victor.

4.7 Display Status Method

This method should iterate over all characters in the game, displaying their name, health, and whether they are the current active character.

Implementation Steps:

1. **Iterate Through Characters**: Go through each character in the game's character array.
2. **Compile Character Details**: For each character, compile their name, health status, and a note if they are the current active character.

3. **Format the Output**: Combine the details into a readable format, as an array of strings.
4. **Return or Log the Output**: Log the compiled status report to the console.