Textual RPG Game

Objective:
Create a simple text-based RPG game system, built using JavaScript ES6 classes, showcasing object-oriented programming concepts such as encapsulation, inheritance, and polymorphism.

Step 1: Character Superclass

Objective

In this step, you will be writing a Character class in JavaScript. This class will represent a game character, and will have properties like name, health, strength, and an inventory of items. It will also have methods for attacking another character, adding an item to the inventory, removing an item from the inventory, and displaying the character's details.

Instructions

1. Define the `Character` class: Start by defining a class named `Character` using the `class` keyword.
2. Define the Constructor: Inside the `Character` class, define a constructor that accepts three parameters: `name`, `health`, and `strength`. These parameters represent the character's name, health points, and strength points, respectively.
3. Define Inventory: In the same constructor, initialize an empty array for the character's inventory. This will store the items that the character has.
4. Define the `attack` method: This method should accept a `target` character as an argument, and subtract the attacking character's `strength` from the target's `health`. It should return a string stating who attacked who and the remaining health of the target character.
5. Define the `addItem` method: This method should accept an `item` as an argument and add it to the character's inventory.
6. Define the `removeItem` method: This method should accept an `item` as an argument and remove it from the character's inventory. You should first find the index of the item in the inventory array, and if it is present (index is not -1), you can remove it using the `splice` method.

7. Define the `displayCharacter` method: This method should return a string that contains the character's name, health points, strength points, and the list of items in the inventory.

Step 2: Player and Enemy Subclasses

Objective

In this step, you will be extending the previously created Character class to create two new classes: Player and Enemy. The Player class will add a level attribute and a method to upgrade the player. The Enemy class will add a type attribute and override the displayCharacter method to include this new attribute.

Instructions

1. Define the `Player` class: Start by defining a class named `Player` that extends `Character` using the `extends` keyword. This means that the `Player` class will inherit all properties and methods of the `Character` class.

2. Define the Constructor: Inside the `Player` class, define a constructor that accepts four parameters: `name`, `health`, `strength`, and `level`. Use the `super` keyword to call the constructor of the parent `Character` class for `name`, `health`, and `strength`. Then, initialize `level`.

3. Define the `upgrade` method: This method should increment the `level` of the player by 1, `health` by 10, and `strength` by 5.

4. Define the `Enemy` class: Similar to the `Player` class, define a class named `Enemy` that also extends `Character`.

5. Define the Constructor: Inside the `Enemy` class, define a constructor that accepts four parameters: `name`, `health`, `strength`, and `type`. Like the `Player` class, use `super` to call the parent class constructor for `name`, `health`, and `strength`. Then, initialize `type`.

6. Override the `displayCharacter` method: This method should call the `displayCharacter` method of the `Character` class using the `super` keyword and then add the `type` information to the string.

Step 3: Item, HealthPotion and StrengthElixir Classes

In this step, you will be creating an `Item` class representing an item in the game. This class will have properties like `name` and `description`, and a `use` method that will apply the item's effect on a game character. You will also be creating two subclasses: `HealthPotion` and `StrengthElixir`, which will override the `use` method to provide specific effects.

Instructions

1. Define the `Item` class: Start by defining a class named `Item` using the `class` keyword.
2. Define the Constructor: Inside the `Item` class, define a constructor that accepts two parameters: `name` and `description`. These parameters represent the item's name and its description, respectively.
3. Define the `use` method: This method should accept a `target` character as an argument, and for the base class, it will only console.log a message stating that the item is being used on the target. Note that this method will be overridden in the subclasses to provide specific functionality.
4. Define the `HealthPotion` class: This class should extend the `Item` class. It will override the `use` method to increase the `health` of the target character by 30.
5. Define the `StrengthElixir` class: This class should also extend the `Item` class. It will override the `use` method to increase the `strength` of the target character by 10.

Step 4: Game Class

Objective

In this exercise, you'll be creating a `Game` class to manage the gameplay. This class will have properties to store the player, enemies, and items, and methods to start and end the game, spawn enemies and items, allow the player to pick up and use items, and allow the player to attack enemies.

Instructions

1. Define the `Game` class: Start by defining a class named `Game` using the `class` keyword.
2. Define the Constructor: Inside the `Game` class, define a constructor that initializes `player` as `null`, `enemies` as an empty array, and `items` as an empty array.

3. Define the `startGame` method: This method should accept a `playerName` as an argument and create a new `Player` object, assigning it to `this.player`.

4. Define the `endGame` method: This method should reset `player`, `enemies`, and `items` to their initial states.

5. Define the `spawnEnemy` method: This method should accept `enemyName`, `enemyHealth`, and `enemyStrength` as arguments, create a new `Enemy` object, and add it to `this.enemies`.

6. Define the `spawnItem` method: This method should accept `itemName` and `itemDescription` as arguments, create a new `Item` (or its subclass) object based on the `itemName`, and add it to `this.items`.

> **Detailed Instructions for the spawnItem method:**
>
> - First, start by defining a new method called `spawnItem` which takes two arguments, `itemName` and `itemDescription`. These arguments represent the name and description of the item that we want to spawn in the game.
> - Within the `spawnItem` method, declare a variable called `item` without assigning it a value. This variable will be used later to store the new item that we will create.
> - Now, we want to create different types of items based on the `itemName`. To do this, set up an if-else conditional structure to check the value of `itemName`.
> - If the `itemName` is exactly equal to the string "Health Potion", create a new `HealthPotion` object using the `new` keyword. Pass `itemName` and `itemDescription` as arguments to the `HealthPotion` constructor. Assign this new `HealthPotion` object to the `item` variable.
> - If the `itemName` is not "Health Potion", but is exactly equal to the string "Strength Elixir", create a new `StrengthElixir` object, again using the `new` keyword. Pass `itemName` and `itemDescription` as arguments to the `StrengthElixir` constructor. Assign this new `StrengthElixir` object to the `item` variable.
> - If `itemName` does not match either "Health Potion" or "Strength Elixir", create a new generic `Item` object. Pass `itemName` and `itemDescription` as arguments to the `Item` constructor. Assign this new `Item` object to the `item` variable. This is our fallback for any other types of items not specifically defined in our code.
> - Now that we have created a new item based on `itemName` and stored it in the `item` variable, we need to add it to our game. Do this by pushing the `item` into the `items` array of our `Game` instance using the `push` method. This represents the spawning of the item in our game world.

7. Define the `playerPickUpItem` method: This method should accept an `item` as an argument, remove it from `this.items`, and add it to the player's inventory.
8. Define the `playerUseItem` method: This method should accept an `item` and a `target` as arguments. If the item is in the player's inventory, it should use the item on the target and remove it from the player's inventory.

9. Define the `playerAttack` method: This method should accept an `enemy` as an argument and retrun the player's `attack` method with the enemy as the target.