CSE101T Assignment 3

Monster Team and Strategic Expedition

Due: 3.12.2025 by 23:59

Deliverables

Submit a single Java file named `Assignment3.java` via **LMS System (Moodle)** by the due date and time specified. Late submissions will be subject to the course's late policy. Submissions via MS Teams **will not be graded.**

https://lms.akdeniz.edu.tr

The Story

Your monster has successfully completed `The Guardian's Trial` and earned the right to embark on new adventures. However, great discoveries cannot be made alone. In this assignment, your monster will lead a team and set out on various exploration missions across a mysterious map. By acquiring teammates, you will encounter challenges along the way, and your decisions will determine the fate of your team. Your goal is to explore all regions on the map and find the hidden treasure.

Program Flow: Form a Team, Explore, Make Decisions (Simulated)

The program must continue as a single, continuous narrative after the monster creation, training, and Guardian trial processes from the previous assignment.

1. **Part 1: Character Creation, Training, and Guardian Trial (Assignment 2 Logic):** The program must include all steps from Assignment 2. The monster is created, trained, and fights against the Stone Guardian. If the monster defeats the Guardian, the logic for Assignment 3 begins. In case of defeat, the program should terminate with an appropriate message (no retry option will be provided).
2. **Part 2: Team Formation (New Logic):** Immediately after defeating the Guardian, your monster will acquire 2 new teammates. These teammates will have different attributes and will complement your monster's shortcomings.
3. **Part 3: Exploration Missions and Decision Making (New Logic):** Once the team is ready, you will embark on exploration missions across a map. In each mission, your team will encounter an event, and strategic decisions will be *simulated*. These simulated decisions can affect the team's HP, AP, or DP.

Your Task - Method and Array Focused Implementation (No Scanner Input)

Your primary objective is to build upon your working Assignment 2 code by adding the Team Formation and Exploration Modules, creating a single `Assignment3.java` file. This assignment specifically focuses on the **effective use of methods and arrays**. You are expected

to implement the provided method signatures, ensuring that these methods perform "serious" manipulations on arrays to achieve their stated purpose. While you may add helper methods, the core flow must be managed through the specified methods.

CRITICAL DIRECTIVE: You MUST NOT use the `Scanner` class anywhere in your `Assignment3.java` file. All user inputs that were previously handled by `Scanner` in Assignment 2 (e.g., monster name, element choice, training days) should be treated as pre-determined values (e.g., hardcoded for testing, or passed as initial parameters to your `main` method for testing purposes). Similarly, all decisions within Part 3 (e.g., region choice, event choices) must be passed as parameters to the relevant methods, simulating predefined choices for testing.

Part 1: Mechanics of Creation, Training, and Trial

This part is identical to Assignment 2, with the crucial modification that **all user inputs must be simulated**. For example, instead of reading `monsterName` from `Scanner`, assume it's "Sparky". Similarly for element, potential, destiny number, and training days. You must integrate your working code, adapting it to this non-interactive input method. The new sections (Part 2 and Part 3) will only activate if your monster **defeats** the Stone Guardian.

Part 2: Mechanics of Team Formation and Management

Upon defeating the Guardian, your monster will be joined by two new teammates. These teammates have fixed statistics and names. The player does not choose these teammates; they automatically join the team.

Data Structure for Team Members

All team members (including your main monster, totaling 3 members) will have their names, current/max HPs, AP, DP, and alive status managed using arrays. These arrays should be defined in the `main` method or at a higher scope and passed as parameters to the methods.

- `String[] teamMemberNames`
- `int[] teamMemberCurrentHP`
- `int[] teamMemberMaxHP`
- `int[] teamMemberAP`
- `int[] teamMemberDP`
- `boolean[] teamMemberAlive`

Team Member Details

- **Your Monster:** The monster created in Part 1 with its final stats after training.
- **Teammate 1: "Swiftclaw"**

  - HP: 80
  - AP: 40
  - DP: 10
  - Characteristic: High attack and speed.

- **Teammate 2: "Ironhide"**
  - HP: 120
  - AP: 20
  - DP: 30
  - Characteristic: High defense and durability.

Required Methods for Team Management

1. **`initializeTeam`**
   - `public static void initializeTeam(String monsterName, int monsterHP, int monsterAP, int monsterDP, String[] teamMemberNames, int[] teamMemberCurrentHP, int[] teamMemberMaxHP, int[] teamMemberAP, int[] teamMemberDP, boolean[] teamMemberAlive)`
   - **Purpose:** Populates the team member arrays according to a fixed structure. It uses the player's monster stats for the first member and adds two predefined companions.
   - **Implementation Details:** Your implementation must place the data at the following specific indices:
     - **Index 0 (Player's Monster):**
       - Use the data from the `monsterName`, `monsterHP`, `monsterAP`, and `monsterDP` parameters.
       - Set both `CurrentHP` and `MaxHP` to the value of `monsterHP`.
       - Set `teamMemberAlive[0]` to `true`.
     - **Index 1 (Companion 1):**
       - Name: `"Swiftclaw"`
       - HP: 80 (Set both Current and Max HP)
       - AP: 40
       - DP: 10
       - Set `teamMemberAlive[1]` to `true`.
     - **Index 2 (Companion 2):**
       - Name: `"Ironhide"`
       - HP: 120 (Set both Current and Max HP)
       - AP: 20
       - DP: 30
       - Set `teamMemberAlive[2]` to `true`.

2. **`calculateTotalTeamStats`**

    - `public static int[] calculateTotalTeamStats(int[] teamMemberCurrentHP, int[] teamMemberAP, int[] teamMemberDP, boolean[] teamMemberAlive)`
    - **Purpose:** Calculates the team's current total statistics by summing the current HP, AP, and DP values *only* for team members whose status in the `teamMemberAlive` array is `true` (i.e., conscious members).
    - **Return Value:** An `int[]` array in the format: `[totalHP, totalAP, totalDP]`.

3. **`displayTeamStats`**

    - `public static void displayTeamStats(String[] teamMemberNames, int[] teamMemberCurrentHP, int[] teamMemberMaxHP, int[] teamMemberAP, int[] teamMemberDP, boolean[] teamMemberAlive, int[] totalTeamStats)`
    - **Purpose:** Prints the details of each team member (name, current HP/Max HP, AP, DP, alive status) and the team's total statistics (HP, AP, DP) in a well-formatted manner. For unconscious members, "Unconscious" should be indicated.

Part 3: Mechanics of Exploration Missions and Decision Making

Once the team is formed, you will embark on a series of exploration missions. The entire exploration process is deterministic. The specific events encountered and the decisions made in response are predefined in the 'main' method's test scenario. The map consists of 3 distinct regions, each with a fixed list of scenarios.

Data Structures for Exploration

- `boolean[] regionsExploredStatus = {false, false, false};` (For Forest, Cave, Ancient Ruins)

Required Methods for Exploration

4. **`allRegionsExplored`**

    `public static boolean allRegionsExplored(boolean[] regionsExploredStatus)`

    - **Purpose:** Checks if all regions have been explored by verifying if all elements in the 'regionsExploredStatus' array are 'true'.
    - **Return Value:** `true` if all regions have been explored, `false` otherwise.

5. **`triggerEvent`**

    - `public static void triggerEvent(int regionIndex, int scenarioIndex, int simulatedChoice, String[] teamMemberNames, int[] teamMemberCurrentHP, int[]`

`teamMemberMaxHP, int[] teamMemberAP, int[] teamMemberDP, boolean[] teamMemberAlive)`

- **Purpose:** Deterministically triggers a specific event. It uses the `regionIndex` and the `scenarioIndex` to execute a specific scenario. It then applies the predefined outcome based on the `simulatedChoice`. The method must first print the event's description and then the outcome.
- **Input Validation:** This method must validate its input parameters. If `scenarioIndex` is a value other than 0 or 1, it **must default to 0**. If `simulatedChoice` is a value other than 1 or 2, it **must default to 1**.

6. **`applyDamageToTeam`**

   - `public static void applyDamageToTeam(int damage, int[] teamMemberCurrentHP, int[] teamMemberDP, boolean[] teamMemberAlive)`
   - **Purpose:** Distributes the specified `damage` among the conscious team members. For this assignment, distribute the damage equally among all conscious members (integer division is acceptable 10 / 3 = 3). If a member's HP drops to 0 or below, set their status in 'teamMemberAlive' to 'false' and their HP to 0.

7. **`healTeam`**

   - `public static void healTeam(int healAmount, int[] teamMemberCurrentHP, int[] teamMemberMaxHP, boolean[] teamMemberAlive)`
   - **Purpose:** Distributes the specified `healAmount` among conscious team members whose HP is not full. For this assignment, distribute the healing equally. A member's HP must not exceed their 'teamMemberMaxHP'. If an unconscious member is healed (HP > 0), their 'teamMemberAlive' status must become 'true'.

8. **`checkTeamStatus`**

   - **Signature:** `public static boolean checkTeamStatus(boolean[] teamMemberAlive)`
   - **Purpose:** Checks if at least one team member is still conscious.
   - **Return Value:** `true` if at least one member is alive, `false` otherwise.

Example Scenario: `applyDamageToTeam` Logic

Assume the method is called with a damage amount of 25: `applyDamageToTeam(25, teamMemberCurrentHP, teamMemberDP, teamMemberAlive);`

Initial State

The table below shows the state of the team members **before** the method is called.

Table 1: State of Team Members **Before** Calling `applyDamageToTeam`

Member	Max HP	Current HP	Status
Member 1	30	25	Alive
Member 2	15	8	Alive
Member 3	20	0	Unconscious

Execution Logic

The method follows these steps:

1. **Identify Eligible Members:** The method first identifies which members can receive damage. The rule is to include only conscious (alive) members.
   - Member 1 (25 HP) is eligible.
   - Member 2 (8 HP) is eligible.
   - Member 3 (0 HP) is **not** eligible because they are already unconscious.

   This results in a total of **2 eligible members**.
2. **Calculate Damage Per Member:** The total damage amount is divided by the number of eligible members using integer division.
   - `25 / 2 = 12`. Each eligible member will take **12 damage**. The remaining 1 point of damage is discarded due to integer division.
3. **Apply Damage and Check Rules:** The damage is applied to each eligible member individually.
   - **For Member 1:** The new HP is `25 - 12 = 13`. Since this is above 0, their status remains 'Alive'. The final HP is **13**.
   - **For Member 2:** The new potential HP would be `8 - 12 = -4`. Since this is less than or equal to 0, two rules are applied:
     (a) The final HP is set to **0** (it cannot be negative).
     (b) Their status in the `teamMemberAlive` array must be set to `false`.
   - **For Member 3:** No damage is applied as they were not eligible. The HP remains **0**.

Final State

The table below shows the state of the team members **after** the method has finished its execution.

Table 2: State of Team Members **After** Calling `applyDamageToTeam`

Member	Max HP	Current HP	Status
Member 1	30	**13**	Alive
Member 2	15	**0**	**Unconscious**
Member 3	20	**0**	Unconscious

Example Scenario: `healTeam` Logic

To clarify how the `healTeam` method should be implemented, let's walk through a specific scenario. This example demonstrates the rules of equal distribution, capping at max HP, and reviving unconscious members.

Assume the method is called with a healing amount of 20: `healTeam(20, teamMemberCurrentHP, teamMemberMaxHP, teamMemberAlive);`

Initial State

The table below shows the state of the team members **before** the method is called.

Table 3: State of Team Members **Before** Calling `healTeam`

Member	Max HP	Current HP	Status
Member 1	14	13	Alive
Member 2	10	10	Alive
Member 3	20	0	Unconscious

Execution Logic

The method follows these steps:

1. **Identify Eligible Members:** The method first identifies which members can receive healing. The rule is to include conscious members with non-full HP and all unconscious members.

   - Member 1 (13/14 HP) is eligible.
   - Member 2 (10/10 HP) is **not** eligible because their HP is full.
   - Member 3 (0/20 HP) is eligible.

   This results in a total of **2 eligible members**.

2. **Calculate Healing Per Member:** The total healing amount is divided by the number of eligible members using integer division.

   - `20 / 2 = 10`. Each eligible member will attempt to receive **10 HP**.

3. **Apply Healing and Check Rules:** The healing is applied to each eligible member individually.

   - **For Member 1:** The new potential HP would be `13 + 10 = 23`. Since this exceeds the Max HP of 14, the final HP is capped at **14**.
   - **For Member 2:** No healing is applied as they were not eligible. The HP remains **10**.
   - **For Member 3:** The new HP is `0 + 10 = 10`. This does not exceed the Max HP of 20. Since the new HP is greater than 0, their status in the `teamMemberAlive` array must be set to `true`. The final HP is **10**.

Final State

The table below shows the state of the team members **after** the method has finished its execution.

Table 4: State of Team Members **After** Calling `healTeam`

Member	Max HP	Current HP	Status
Member 1	14	**14**	Alive
Member 2	10	**10**	Alive
Member 3	20	**10**	**Alive**

Deterministic Scenario and Choice List

Your ‘triggerEvent‘ method MUST implement the following logic exactly.

Region Index 0: The Murky Forest

- **Scenario Index 0: Wild Monster Ambush**
  - **`simulatedChoice = 1` (Fight):** The team takes damage equal to ‘(40 - Total Team DP)‘ (minimum 5 damage). Distribute with ‘applyDamageToTeam‘. The AP of the conscious member with the lowest AP is permanently increased by 5.
  - **`simulatedChoice = 2` (Distract and Flee):** The attempt to flee **fails**. The team takes 15 damage, distributed with ‘applyDamageToTeam‘.
- **Scenario Index 1: Whispering Grove**
  - **`simulatedChoice = 1` (Meditate):** Call ‘healTeam‘ with a value of 20 HP. The DP of the conscious member with the highest DP is permanently increased by 3.
  - **`simulatedChoice = 2` (Ignore):** No stats change.

Region Index 1: The Crystal Caves

- **Scenario Index 0: Unstable Ceiling**
  - **`simulatedChoice = 1` (Run Through):** The team takes a total of 30 damage, distributed with ‘applyDamageToTeam‘.
  - **`simulatedChoice = 2` (Take Cover):** The attempt to take cover **fails**. The conscious team member with the lowest current HP takes 10 damage.
- **Scenario Index 1: Glowing Crystal Vein**
  - **`simulatedChoice = 1` (Harvest for Power):** The AP of all conscious team members is permanently increased by 5.
  - **`simulatedChoice = 2` (Harvest for Durability):** The Maximum HP and Current HP of all conscious team members are permanently increased by 10.

Region Index 2: The Ancient Ruins

- **Scenario Index 0: Ancient Mechanism**
  - **`simulatedChoice = 1` (Activate):** The mechanism is a **trap**. All conscious members take 20 damage, distributed with ‘applyDamageToTeam‘.
  - **`simulatedChoice = 2` (Walk Around):** No stats change.
- **Scenario Index 1: Sacrificial Altar**
  - **`simulatedChoice = 1` (Make a Sacrifice):** The current HP of all conscious members is reduced by 25% (integer division, minimum HP is 1). In return, the AP and DP of all conscious members are permanently increased by 5.
  - **`simulatedChoice = 2` (Ignore):** No stats change.

Victory Condition

The hidden treasure is found, and a "VICTORY!" message along with the final team statistics and surviving members is displayed when `allRegionsExplored` returns `true`.

Defeat Condition

If `checkTeamStatus` returns `false` at any point (meaning all team members are unconscious), the mission fails, and a "DEFEAT!" message is displayed, terminating the program.

Complete Example Scenario: Part 2 & Part 3

This section details a full run-through of the program, starting from the moment your monster defeats the Stone Guardian. This demonstrates how the methods from Part 2 and Part 3 work together to create a continuous adventure.

Initial State (Post-Assignment 2)

Let’s assume the monster created and trained in Part 1 and 2 is named **"Pyre"** and has the following final stats after defeating the Guardian:

- **HP:** 85
- **AP:** 45
- **DP:** 25

Part 2: Team Formation

The program now transitions to Part 2. The first step is to form the team.

Console Output for Part 2

```
VICTORY! The Guardian deems your monster worthy.
The path to new adventures is now open!

Two skilled adventurers, Swiftclaw and Ironhide, are impressed
by Pyre's strength and decide to join the team!

--- TEAM ROSTER INITIALIZED ---
```

Behind the Scenes

1. The `initializeTeam` method is called with Pyre's stats. It populates all the team member arrays.
2. The `calculateTotalTeamStats` method is called to get the team's combined power.
3. The `displayTeamStats` method is called to print the initial roster.

Console Output of Team Stats

```
+------------------+---------+----+----+-------------+
| Member           | HP      | AP | DP | Status      |
+------------------+---------+----+----+-------------+
| Pyre             | 85/85   | 45 | 25 | Alive       |
| Swiftclaw        | 80/80   | 40 | 10 | Alive       |
| Ironhide         | 120/120 | 20 | 30 | Alive       |
+------------------+---------+----+----+-------------+
| TOTAL STATS      | HP: 285 | AP: 105 | DP: 65      |
+------------------+---------+----+----+-------------+
```

Part 3: The Exploration Begins

The program now enters the main exploration loop defined in the ‘main‘ method. It will execute the following predefined test sequence:

- **Step 1:** Region Index 0 (Forest), Scenario 0 (Ambush), Choice 1 (Fight).
- **Step 2:** Region Index 1 (Caves), Scenario 1 (Crystal), Choice 2 (Harvest Durability).
- **Step 3:** Region Index 2 (Ruins), Scenario 0 (Mechanism), Choice 1 (Activate).

Step 1: The Murky Forest

```
--- Exploring Region Index: 0, Scenario Index: 0 ---
```

Logic: The `triggerEvent` method is called with ‘regionIndex=0‘, ‘scenarioIndex=0‘, and ‘simulatedChoice=1‘.

- The method executes the "Wild Monster Ambush" scenario with the "Fight" choice.

- Damage calculation: ‘(40 - Total Team DP)‘ -> ‘40 - 65 = -25‘. Minimum damage is 5.
- `applyDamageToTeam` is called with 5 damage. With 3 alive members, ‘5 / 3 = 1‘. Each member takes 1 damage.
- Bonus: The AP of the member with the lowest AP (Ironhide, with 20 AP) is increased by 5. Ironhide’s new AP is 25.

Console Output for Step 1

```
Event: Your team is ambushed by wild monsters.
Outcome: The team fights back bravely! They take 5 damage in the
skirmish. Ironhide’s resolve is hardened, increasing its AP by 5.

+------------------+---------+----+----+-------------+
| Member           | HP      | AP | DP | Status      |
+------------------+---------+----+----+-------------+
| Pyre             | 84/85   | 45 | 25 | Alive       |
| Swiftclaw        | 79/80   | 40 | 10 | Alive       |
| Ironhide         | 119/120 | 25 | 30 | Alive       |
+------------------+---------+----+----+-------------+
| TOTAL STATS      | HP: 282 | AP: 110 | DP: 65      |
+------------------+---------+----+----+-------------+
```

Step 2: The Crystal Caves

```
--- Exploring Region Index: 1, Scenario Index: 1 ---
```

Logic: The `triggerEvent` method is called with ‘regionIndex=1‘, ‘scenarioIndex=1‘, and ‘simulatedChoice=2‘.

- The method executes the "Glowing Crystal Vein" scenario with the "Harvest for Durability" choice.
- All 3 alive members have their Max HP and Current HP increased by 10.

Console Output for Step 2

```
Event: You discover a glowing crystal vein in the cave wall.
Outcome: The team harvests the crystals, feeling a surge of vitality.
Their maximum and current HP increase!

+------------------+---------+----+----+-------------+
| Member           | HP      | AP | DP | Status      |
+------------------+---------+----+----+-------------+
| Pyre             | 94/95   | 45 | 25 | Alive       |
| Swiftclaw        | 89/90   | 40 | 10 | Alive       |
| Ironhide         | 129/130 | 25 | 30 | Alive       |
+------------------+---------+----+----+-------------+
| TOTAL STATS      | HP: 312 | AP: 110 | DP: 65      |
+------------------+---------+----+----+-------------+
```

Step 3: The Ancient Ruins

```
--- Exploring Region Index: 2, Scenario Index: 0 ---
```

Logic: The `triggerEvent` method is called with ‘regionIndex=2‘, ‘scenarioIndex=0‘, and ‘simulatedChoice=1‘.

- The method executes the "Ancient Mechanism" scenario with the "Activate" choice.
- The mechanism is a trap. The team takes 20 damage.
- `applyDamageToTeam` is called with 20 damage. With 3 alive members, ‘20 / 3 = 6‘. Each member takes 6 damage.

Console Output for Step 3

```
Event: You find an ancient mechanism on a pressure plate.
Outcome: It's a trap! The mechanism releases a burst of negative
energy, damaging the team.

+------------------+---------+----+----+-------------+
| Member           | HP      | AP | DP | Status      |
+------------------+---------+----+----+-------------+
| Pyre             | 88/95   | 45 | 25 | Alive       |
| Swiftclaw        | 83/90   | 40 | 10 | Alive       |
| Ironhide         | 123/130 | 25 | 30 | Alive       |
+------------------+---------+----+----+-------------+
| TOTAL STATS      | HP: 294 | AP: 110 | DP: 65      |
+------------------+---------+----+----+-------------+
```

Conclusion of the Mission

The ‘for‘ loop in the ‘main‘ method completes. The program then checks the final conditions.

- `checkTeamStatus` returns ‘true‘ (all members are alive).
- `allRegionsExplored` returns ‘true‘ (all three regions were visited).

The victory condition is met.

Final Console Output

```
VICTORY! All regions explored successfully!
```