Battle Arena – A Java OOP Project

Index

1. Introduction

Battle Arena is a turn-based combat simulation game developed in Java. The game models a system where heroes automatically battle AI-driven enemies across multiple levels with varying challenges like traps, obstacles, and dynamic bonuses. The project showcases the complete application of object-oriented programming (OOP) principles through carefully structured, modular code.

2. Libraries and Frameworks

The game is developed in Java 17 using IntelliJ IDEA. Maven is used for dependency and project management. Only Java standard libraries are used, including java.util for collections and randomness. No third-party frameworks were required.

3. Bottom-Up Design Overview

The system starts with classes such as Hero, Enemy, Weapon, Bonus, and Obstacle. Interfaces define common behavior (IBonus, IEnemy), and abstract classes such as Enemy provide shared logic. These classes are assembled into a Level, and levels are managed by LevelManager. The game runs without player input—heroes and enemies interact based on coded logic.

4. Class Interactions

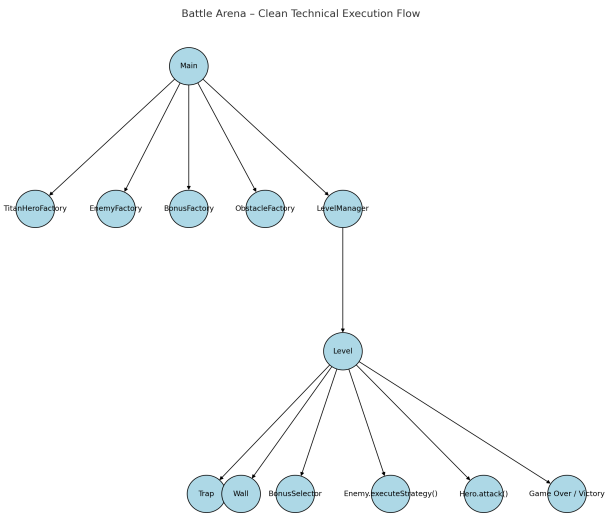
Entities interact through method calls, interface references, and modular factories. Each Hero has a Weapon (composition), and Level instances contain lists of Enemies, Bonuses, and Obstacles. Enemies are assigned behaviors using the Strategy Pattern (AttackStrategy), allowing flexible combat logic. BonusSelector dynamically chooses bonuses based on hero state, and ObstacleFactory adjusts trap and wall difficulty. All gameplay is managed sequentially by LevelManager.

(Diagram: Hero → Weapon, Hero → IBonus, Enemy → AttackStrategy, Level → Enemy/Bonus/Obstacle, Enemy ← Zombie/Ogre, Enemy ← Strategy)

- Flow Diagram

What it shows:

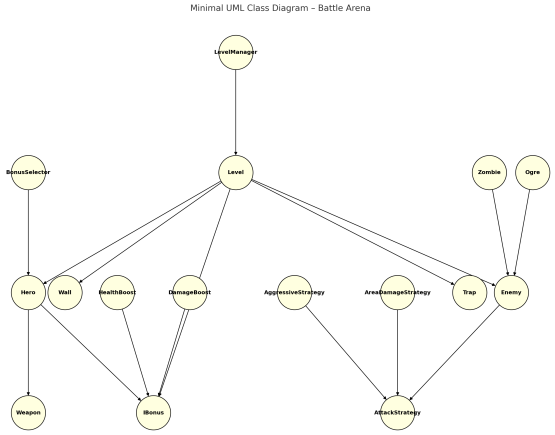
Main calls factories → builds all level content
LevelManager controls game flow
Level handles everything in order:
Trap → Wall → Bonus → Enemy → Hero → End



- UML Class Diagram

  What it shows:

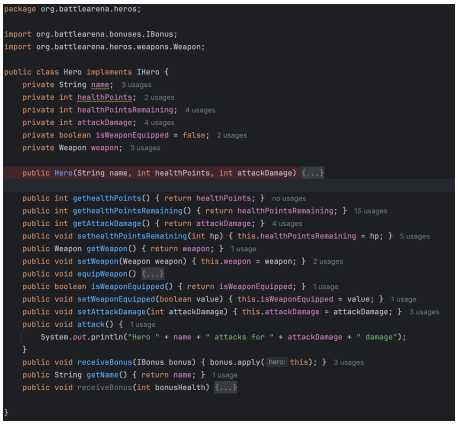
  Inheritance: Zombie and Ogre → Enemy
  Strategy use: Enemy→AttackStrategy→Aggressive/AreaDamageStrategy
  Bonuses: Hero uses IBonus → HealthBoost, DamageBoost
  Composition: Level has all game components
  LevelManager manages Level



5.1 Encapsulation

Definition: Encapsulation is the practice of bundling data with methods that operate on that data, while restricting direct access to internal variables.

In Battle Arena: All fields in core classes are declared private and are accessed/modified only via public getters and setters. This ensures data safety and separation of internal representation.

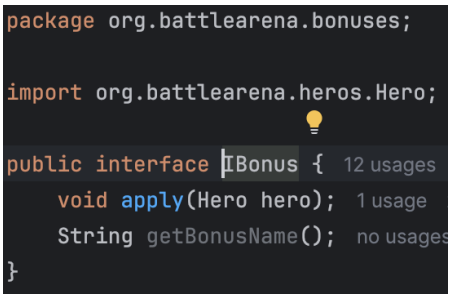


```
package org.battlearena.heros;

import org.battlearena.bonuses.IBonus;
import org.battlearena.heros.weapons.Weapon;

public class Hero implements IHero {
    private String name;  3 usages
    private int healthPoints;  2 usages
    private int healthPointsRemaining;  4 usages
    private int attackDamage;  4 usages
    private boolean isWeaponEquipped = false;  2 usages
    private Weapon weapon;  3 usages

    public Hero(String name, int healthPoints, int attackDamage) {...}

    public int gethealthPoints() { return healthPoints; }  no usages
    public int gethealthPointsRemaining() { return healthPointsRemaining; }  15 usages
    public int getAttackDamage() { return attackDamage; }  4 usages
    public void sethealthPointsRemaining(int hp) { this.healthPointsRemaining = hp; }  5 usages
    public Weapon getWeapon() { return weapon; }  1 usage
    public void setWeapon(Weapon weapon) { this.weapon = weapon; }  2 usages
    public void equipWeapon() {...}
    public boolean isWeaponEquipped() { return isWeaponEquipped; }  1 usage
    public void setWeaponEquipped(boolean value) { this.isWeaponEquipped = value; }  1 usage
    public void setAttackDamage(int attackDamage) { this.attackDamage = attackDamage; }  3 usages
    public void attack() {  1 usage
        System.out.println("Hero " + name + " attacks for " + attackDamage + " damage");
    }
    public void receiveBonus(IBonus bonus) { bonus.apply( hero: this); }  3 usages
    public String getName() { return name; }  1 usage
    public void receiveBonus(int bonusHealth) {...}
}
```

5.2 Information Hiding

Definition: Information hiding prevents external classes from relying on the internal workings of a class.

In Battle Arena: Heroes apply bonuses without knowing their logic, and obstacles are triggered through interfaces.



```
package org.battlearena.bonuses;

import org.battlearena.heros.Hero;

public interface IBonus {  12 usages
    void apply(Hero hero);  1 usage
    String getBonusName();  no usages
}
```

5.3 Polymorphism

Polymorphism is shown through overriding methods like specialAttack(), and by processing objects through interfaces like List<IBonus> or List<IEnemy>. The correct version of apply() or attack() is resolved at runtime.

```
List<IBonus> bonuses = List.of(
        new HealthBoost( boostAmount: 1),
        new DamageBoost( boostAmount: 1)
);
```

```
@Override 1 usage
public void specialAttack() {
    if (Math.random() < 0.5) {
        setHealthPointsRemaining(getHealthPointsRemaining() + 2);
        System.out.println("🧟‍♂️ Zombie regenerated two Health Points");
    }
}
```

Additionally, polymorphism is achieved using a Strategy Pattern that allows enemies to determine their behavior dynamically. Each Enemy has a reference to an AttackStrategy interface, which is implemented by classes such as AggressiveStrategy, DefensiveStrategy, and AreaDamageStrategy. These enable polymorphic execution of different attack() logics without modifying the enemy class itself.

```
private AttackStrategy strategy;

public void setStrategy(AttackStrategy strategy) {
    this.strategy = strategy;
}
```

```
public void executeStrategy(List<Hero> heroes) { 1 usage
    if (strategy != null) {
        strategy.execute( enemy: this, heroes);
    } else {
        Hero fallback = heroes.stream().filter(h -> h.gethealthPointsRemaining() > 0).findFirst().orElse( other: null);
        if (fallback != null) attack(fallback);
    }
}
```

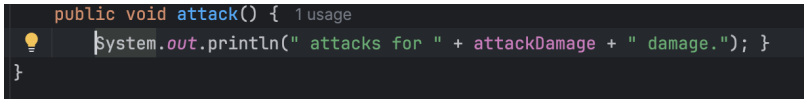
5.4 Composition

Hero has a Weapon (object in a field), and Level has lists of Enemies, Bonuses, and Obstacles. These demonstrate has-a relationships central to composition.

```
private Weapon weapon;  3 usages

public void equipWeapon() {  2 usages
    if (getWeapon() != null && !isWeaponEquipped()) {
        setAttackDamage(getAttackDamage() + weapon.getAttackIncrease());
        setWeaponEquipped(true);
    }
}
```

5.5 Reuse

Shared behaviors such as attack() and setHealthPointsRemaining() are defined in Enemy and reused in Zombie and Ogre. Similarly, bonus logic is reused through IBonus interface implementations.

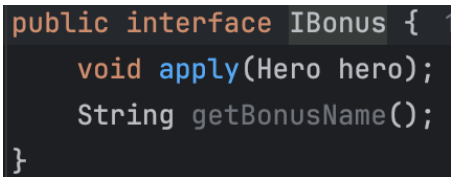


5.6 Abstraction

Definition: Abstraction hides the implementation details while exposing a clean API.

In Battle Arena:

- Interfaces like IBonus, IObstacle and abstract class Enemy define contracts for implementation.

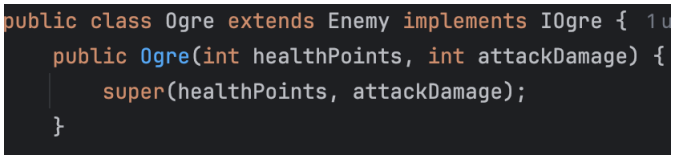


5.7 Inheritance

Definition: Inheritance allows one class to acquire properties and methods from another.

In Battle Arena:

- Enemy is inherited by Zombie and Ogre
- specialAttack() is overridden in each subtype



5.8 Subtyping

Code uses interfaces as types:
List<IBonus> bonuses = List.of(new HealthBoost(), new DamageBoost());
Subtypes are interchangeable through interface references.

```
List<IBonus> bonuses = List.of(
        new HealthBoost( boostAmount: 1),
        new DamageBoost( boostAmount: 1)
);

List<IObstacle> obstacles = List.of(
        new Trap( damage: 2),
        new Wall( blockAmount: 1)
);
```

5.9 Exception Handling

Definition: Exception handling allows programs to catch and deal with errors gracefully. CharacterDeadException is thrown when a hero's HP drops to 0. This exception is caught in LevelManager to stop the game and display an appropriate message.

```
public class CharacterDeadException extends Exception {
    public CharacterDeadException(String message) {  1 usa
        super(message);
    }
}
```

5.10 Strategy Pattern

The Strategy Pattern is used to enable flexible and interchangeable enemy behavior. Each enemy can be assigned a strategy implementing AttackStrategy, such as:

- AggressiveStrategy: targets the strongest hero
- DefensiveStrategy: targets the weakest
- AreaDamageStrategy: targets all heroes

```
public class AreaDamageStrategy implements AttackStrategy {
    @Override  1 usage
    public void execute(Enemy enemy, List<Hero> heroes) {
        enemy.attack(heroes);
    }
}
```

```java
public class DefensiveStrategy implements AttackStrategy { 2 usages
    @Override 1 usage
    public void execute(Enemy enemy, List<Hero> heroes) {
        Hero target = heroes.stream() Stream<Hero>
                .filter(h -> h.gethealthPointsRemaining() > 0)
                .min(Comparator.comparingInt(Hero::getAttackDamage)) Optional<Hero>
                .orElse( other: null);
        if (target != null) {
            enemy.attack(target);
        }
    }
}
```

```java
public class AggressiveStrategy implements AttackStrategy { 4 usages
    @Override 1 usage
    public void execute(Enemy enemy, List<Hero> heroes) {
        Hero target = heroes.stream() Stream<Hero>
                .filter(h -> h.gethealthPointsRemaining() > 0)
                .max(Comparator.comparingInt(Hero::getAttackDamage)) Optional<Hero>
                .orElse( other: null);
        if (target != null) {
            enemy.attack(target);
        }
    }
}
```

5.11 Factory Pattern

The Factory Pattern is used extensively throughout the project to encapsulate the creation logic of enemies, heroes, bonuses, and obstacles. This design improves modularity, supports the Open/Closed Principle, and isolates complexity from the Main class.

Multiple factory classes were implemented, including:

- EnemyFactory: generates enemies based on difficulty, assigning strategies dynamically.
- BonusFactory: selects bonus types and even includes randomized penalties in HARD mode.
- ObstacleFactory: creates traps and walls with scalable impact.
- TitanHeroFactory: provides the correct team composition (Titan or dual heroes) according to game difficulty.

```java
public class EnemyFactory {  5 usages
    public static List<Enemy> createEnemies(Difficulty difficulty) {  4 usages
        List<Enemy> enemies = new ArrayList<>();

        switch (difficulty) {
            case EASY -> {
                Zombie zombie = new Zombie( healthPoints: 12,  attackDamage: 1);
                zombie.setStrategy(new AggressiveStrategy());
                Ogre ogre = new Ogre( healthPoints: 14,  attackDamage: 2);
                ogre.setStrategy(new AreaDamageStrategy());
                enemies.add(zombie);
                enemies.add(ogre);
            }
            case MEDIUM -> {
                Zombie zombie = new Zombie( healthPoints: 16,  attackDamage: 2);
                zombie.setStrategy(new AggressiveStrategy());
                Ogre ogre = new Ogre( healthPoints: 20,  attackDamage: 3);
                ogre.setStrategy(new AreaDamageStrategy());
                enemies.add(zombie);
                enemies.add(ogre);
            }
            case HARD -> {
                Zombie zombie = new Zombie( healthPoints: 20,  attackDamage: 3);
                zombie.setStrategy(new AggressiveStrategy());
                Ogre ogre = new Ogre( healthPoints: 24,  attackDamage: 5);
                ogre.setStrategy(new AreaDamageStrategy());
                enemies.add(zombie);
                enemies.add(ogre);
            }
        }

        return enemies;
    }
}
```

```java
public class ObstacleFactory {  5 usages
    public static List<IObstacle> createObstacles(Difficulty difficulty) {
        List<IObstacle> obstacles = new ArrayList<>();

        switch (difficulty) {
            case EASY -> {
                obstacles.add(new Trap( damage: 1));
                obstacles.add(new Wall( blockAmount: 1));
            }
            case MEDIUM -> {
                obstacles.add(new Trap( damage: 2));
                obstacles.add(new Wall( blockAmount: 2));
            }
            case HARD -> {
                obstacles.add(new Trap( damage: 4));
                obstacles.add(new Wall( blockAmount: 3));
            }
        }

        return obstacles;
    }
}
```

```java
public class BonusFactory {  5 usages
    public static List<IBonus> createBonuses(Difficulty difficulty) {
        List<IBonus> bonuses = new ArrayList<>();
        Random rand = new Random();

        switch (difficulty) {
            case EASY -> {
                bonuses.add(new HealthBoost( boostAmount: 2));
                bonuses.add(new DamageBoost( boostAmount: 2));
            }
            case MEDIUM -> {
                bonuses.add(new HealthBoost( boostAmount: 1));
                bonuses.add(new DamageBoost( boostAmount: 1));
            }
            case HARD -> {
                if (rand.nextDouble() < 0.7) {
                    bonuses.add(new DamageBoost( boostAmount: 1));

                }
            }
        }

        return bonuses;
    }
}
```

```java
public class TitanHeroFactory {  2 usages
    public static List<Hero> createHeroes(Difficulty difficulty) {  1 usage
        List<Hero> heroes = new ArrayList<>();

        switch (difficulty) {
            case EASY -> {
                Hero hero1 = new Hero( name: "Ares",  healthPoints: 18,  attackDamage: 2);
                Hero hero2 = new Hero( name: "Luna",  healthPoints: 17,  attackDamage: 2);
                hero1.setWeapon(new Weapon( weaponType: "Sword",  attackIncrease: 2));
                hero2.setWeapon(new Weapon( weaponType: "Axe",  attackIncrease: 2));
                hero1.equipWeapon();
                hero2.equipWeapon();
                heroes.add(hero1);
                heroes.add(hero2);
            }
            case MEDIUM -> {
                Hero hero1 = new Hero( name: "Ares",  healthPoints: 20,  attackDamage: 3);
                Hero hero2 = new Hero( name: "Luna",  healthPoints: 18,  attackDamage: 3);
                hero1.setWeapon(new Weapon( weaponType: "Sword",  attackIncrease: 2));
                hero2.setWeapon(new Weapon( weaponType: "Axe",  attackIncrease: 2));
                hero1.equipWeapon();
                hero2.equipWeapon();
                heroes.add(hero1);
                heroes.add(hero2);
            }
            case HARD -> {
                Hero titan = new Hero( name: "Titan",  healthPoints: 33,  attackDamage: 5);
                titan.setWeapon(new Weapon( weaponType: "GodBlade",  attackIncrease: 3));
                titan.equipWeapon();
                heroes.add(titan);
            }
        }

        return heroes;
    }
```

6.1 Multiple Players

The system supports multiple heroes, each with independent state and behavior. They participate in a shared battle loop and are managed using List<Hero>. While there is no manual input, this simulates local multiplayer logic where all heroes act in turn.

```
Hero hero1 = new Hero( name: "Ares",  healthPoints: 18,  attackDamage: 2);
Hero hero2 = new Hero( name: "Luna",  healthPoints: 17,  attackDamage: 2);
```

```
List<Hero> heroes = List.of(hero1, hero2);

for (Hero hero : heroes) {
    for (IObstacle obstacle : obstacles) {
        obstacle.trigger(hero);
    }

    System.out.println("🎁 Applying predefined bonuses...");
    for (IBonus bonus : bonuses) {
        if (hero.gethealthPointsRemaining() > 0) {
            hero.receiveBonus(bonus);
        }
    }
```

```
for (Hero hero : heroes) {
    if (hero.gethealthPointsRemaining() > 0 && enemy.getHealthPointsRemaining() > 0) {
        hero.attack();
        enemy.setHealthPointsRemaining(
                enemy.getHealthPointsRemaining() - hero.getAttackDamage()
        );
    }
}
```

6.2 AI-Assisted Gameplay

All combat actions are automatically performed through predefined AI logic:

- Heroes auto-attack enemies
- BonusSelector uses the hero's condition to determine bonus type
- Critical hits and special attacks are triggered probabilistically
- Enemies use assigned combat strategies through the Strategy Pattern:

    - AggressiveStrategy: attacks the strongest hero
    - DefensiveStrategy: attacks the weakest hero
    - AreaDamageStrategy: attacks all living heroes simultaneously

```
for (Enemy enemy : enemies) {
    if (enemy.getHealthPointsRemaining() > 0) {
        enemy.executeStrategy(heroes);
    }
}
```

6.3 Environmental Obstacles

Each level includes environmental elements that influence gameplay difficulty through damage or penalty. Obstacles are dynamically generated using ObstacleFactory based on the current difficulty level.

There are two main types of obstacles:

- Trap

```
@Override 1 usage
public void trigger(Hero hero) {
    hero.sethealthPointsRemaining(hero.gethealthPointsRemaining() - damage);
    System.out.println("⚠ Trap triggered! -" + damage + " HP");
}
```

- Wall

```
@Override 1 usage
public void trigger(Hero hero) {
    int newDamage = hero.getAttackDamage() - blockAmount;
    if (newDamage < 0) newDamage = 0;
    hero.setAttackDamage(newDamage);
    System.out.println(" Wall blocks " + blockAmount + " attack damage from hero!");
}
```

The strength of each obstacle is determined by the difficulty setting:

- In **EASY**: mild effects (-1 HP, block 1 ATK)
- In **HARD**: severe effects (-4 HP, block 3 ATK)

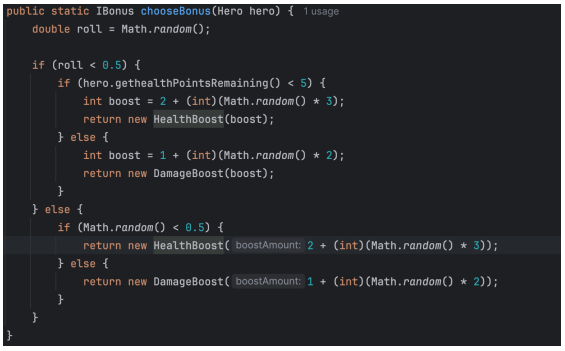
```
public enum Difficulty {
    EASY,  4 usages
    MEDIUM,  4 usages
    HARD  5 usages
}
```

6.4 Dynamic Bonuses

Heroes receive both:

- Predefined bonuses (+1 HP, +1 ATK)
- Smart bonuses, chosen dynamically based on their current health

♦ BonusSelector.java:



```
public static IBonus chooseBonus(Hero hero) {  1 usage
    double roll = Math.random();

    if (roll < 0.5) {
        if (hero.gethealthPointsRemaining() < 5) {
            int boost = 2 + (int)(Math.random() * 3);
            return new HealthBoost(boost);
        } else {
            int boost = 1 + (int)(Math.random() * 2);
            return new DamageBoost(boost);
        }
    } else {
        if (Math.random() < 0.5) {
            return new HealthBoost( boostAmount: 2 + (int)(Math.random() * 3));
        } else {
            return new DamageBoost( boostAmount: 1 + (int)(Math.random() * 2));
        }
    }
}
```

♦ Level.java:

```
for (Hero hero : heroes) {
    for (IObstacle obstacle : obstacles) {
        obstacle.trigger(hero);
    }

    System.out.println("\uD83C\uDF81 Applying predefined bonuses...");
    for (IBonus bonus : bonuses) {
        if (hero.gethealthPointsRemaining() > 0) {
            hero.receiveBonus(bonus);
        }
    }

    System.out.println("\uD83E\uDEA0 Applying smart bonus based on hero state...");
    if (hero.gethealthPointsRemaining() > 0) {
        IBonus selected = BonusSelector.chooseBonus(hero);
        hero.receiveBonus(selected);
    }
}
```

6.5 Level Progression System

Levels are modularly created using the Level class and managed by LevelManager. Each level may have its own configuration of:

- Enemies

- Obstacles
- Bonuses
- Difficulty level (EASY, MEDIUM, HARD)

The hero team composition dynamically changes based on difficulty:

- In EASY and MEDIUM: two heroes (Ares and Luna)
- In HARD: a single powerful hero (Titan)

♦ Main.java

```
Difficulty difficulty = Difficulty.HARD;
```

```
Level level1 = new Level( levelNumber: 1,
        EnemyFactory.createEnemies(difficulty),
        BonusFactory.createBonuses(difficulty),
        ObstacleFactory.createObstacles(difficulty),
        difficulty);
Level level2 = new Level( levelNumber: 2,
        EnemyFactory.createEnemies(difficulty),
        BonusFactory.createBonuses(difficulty),
        ObstacleFactory.createObstacles(difficulty),
        difficulty);
Level level3 = new Level( levelNumber: 3,
        EnemyFactory.createEnemies(difficulty),
        BonusFactory.createBonuses(difficulty),
        ObstacleFactory.createObstacles(difficulty),
        difficulty);

LevelManager manager = new LevelManager(List.of(level1, level2, level3));
```

♦ LevelManager.java

```
public void runLevels(List<Hero> heroes) {  1 usage
    System.out.println(" Starting Game with " + levels.size() + " levels\n");

    for (Level level : levels) {
        try {
            level.startLevel(heroes);
        } catch (CharacterDeadException e) {
            System.out.println("danger!!" + e.getMessage());
            System.out.println("Level " + level.getLevelNumber() + "Game Over.");
            return;
        }

        if (allHeroesDefeated(heroes)) {
            System.out.println("All heroes have been defeated. Game over.");
            return;
        }
    }
```

6.6 Exception Handling and Game Logic Enforcement

The game defines a custom CharacterDeadException, which is thrown when a hero dies. This ensures:

- Immediate termination of the level
- Clean and safe game-over state
- Clear separation of gameplay and control logic

♦ CharacterDeadException.java:

```
public class CharacterDeadException extends Exception {
    public CharacterDeadException(String message) {  1 usa
        super(message);
    }
}
```

♦ Thrown in Level.java:

```
if (targetHero.gethealthPointsRemaining() <= 0) {
    throw new CharacterDeadException(targetHero.getName() + " has fallen in battle!");
}
```

♦ Caught in LevelManager.java:

```
try {
    level.startLevel(heroes);
} catch (CharacterDeadException e) {
    System.out.println("danger!!" + e.getMessage());
    System.out.println("Level " + level.getLevelNumber() + "Game Over.");
    return;
}
```

7. Conclusion

Battle Arena is a complete implementation of object-oriented programming principles. It demonstrates the power of encapsulation, abstraction, and polymorphism in building flexible, extensible systems.Difficulty scaling and dynamic team formation via TitanHeroFactory support gameplay variety and clean code separation.