CS102 – Algorithms and Programming II
Programming Assignment 3
Fall 2023

ATTENTION:

- Compress all of the Java program source files (.java) files into a single zip file.
- The name of the zip file should follow the below convention:
  CS102_Sec1_Asgn3_YourSurname_YourName.zip
- Replace the variables “Sec1”, “YourSurname” and “YourName” with your actual section, surname, and name.
- You may ask questions on Moodle and during your section’s lab.
- Upload the above zip file to Moodle by the deadline (if not, significant points will be taken off). You will get a chance to update and improve your solution by consulting with the TAs and tutors during your section’s lab.

GRADING WARNING:

- Please read the grading criteria provided on Moodle. The work must be done individually. Code sharing is strictly forbidden. We are using sophisticated tools to check the code similarities. The Honor Code specifies what you can and cannot do. Breaking the rules will result in disciplinary action.

7 Unit Fantasy Battle

For this assignment, you will implement a fantasy battle game where the player duels with the computer opponent using a random set of 7 units. At each turn, the player chooses one of the units (that is not dead yet) to send it to the arena. The computer also chooses one randomly. The two units battle at the arena; then, they return to their owner’s hand, dead or alive. Any unit that kills an enemy unit will increase its level by one. The game continues until all units of one side are dead (a tie is also possible).

This lab is about **inheritance and polymorphism**, so you are expected to utilize them in your implementation. First of all, implement an abstract Unit class that includes the following information (we will use classes that extend this Unit class to represent actual units):

- **String name:** Name of this unit. The subclasses will determine this information; it can be Warrior, Archer, Healer, Rogue, Wizard, Bard, or Necromancer.
- **int health:** Current health of this unit. If the unit’s health becomes zero or less, that unit is assumed to be dead. Different effects can heal units; the current health of a unit cannot be more than its maximum health. The first time the unit’s health becomes zero or less, the “isDead” variable of the unit is set to true.
- **int level:** Current level of this unit. Unit level starts from 1 and has no upper bound. Certain effects can lower unit levels; in this case, the level cannot be less than 1.
- **boolean isDead:** Keeps if this unit is alive (false) or dead (true).

Ensure these variables are not private to be able to use them in the subclasses.

Each unit should contain the following abstract methods to be implemented in the classes that extend the Unit class (how they are calculated or function will differ based on the unit, that is why we need to implement them in the subclasses):

- **int getAttack():** Returns the attack points of the unit, this amount will damage the opponent when unit attacks.
- **int getMaxHealth():** Returns the upper boundary for the health of this unit, used when healing or reviving as the upper boundary of the health.
- **void firstPhase(Unit arenaOpponent, ArrayList<Unit> allyWaiting, ArrayList<Unit> enemyWaiting):** Does the actions of the first phase of the battle; the implementation will differ based on the subclass. During any phase, we have access to the both arena units (this class instance and arenaOppenent) and the waiting units of both sides.
- **void secondPhase(Unit arenaOpponent, ArrayList<Unit> allyWaiting, ArrayList<Unit> enemyWaiting):** Does the actions of the second phase of the battle; the implementation will differ based on the subclass.
- **void thirdPhase(Unit arenaOpponent, ArrayList<Unit> allyWaiting, ArrayList<Unit> enemyWaiting):** Does the actions of the third phase of the battle; the implementation will differ based on the subclass.

The following methods will be implemented in the Unit class; as they will not differ based on different units, subclasses will use these methods from their parent class:

- **void damage(int damageAmount):** Subtracts the given damageAmount from this unit's current health. If this damage kills the unit, you should set its isDead variable to true. Any unit that is dead cannot continue to fight; however, certain units can revive dead units, which allows them to continue the game.
- **void increaseLevel():** Increases the level of this unit by 1. Any unit that successfully kills its arena opponent has its level increased. If both sides kill each other, nobody levels up. Certain units can attack the waiting units; if an enemy unit is killed in such a way, the attacking unit still levels up. Leveling up does not cause the unit to heal, it only increases its maximum health and attack points.
- **void decreaseLevel():** Decreases the level of this unit by 1, but the level of the unit cannot be less than 1. This method should also set the unit health to the new maximum health if the new boundary is smaller than the current health. If the current health is already smaller than the new maximum health, no adjustment is needed.
- **void revive():** Only dead units can be revived. Revived units lose 1 level (but their level cannot be less than 1), and their current health is set to their maximum health. Do not forget to set the unit's isDead status to false in this case.
- **void heal(int healAmount):** Increases the health of this unit by the given amount; the unit's health cannot exceed its maximum health.
- **String getInfo():** Returns the string that contains the name, current level, current health, maximum health, and attack points of the Unit. You will use this method while listing the units of the player.

When the game starts, both sides will have 7 random units, all at level 1 and full health. The units that are not in the arena are called waiting units; these are, in a sense, the hand of cards of the player and computer. You will list the units of the player, and after each side chooses a unit to send to the arena, the units battle. The battle consists of three phases. Certain units do different activities at different phases, and units may not do anything at certain phases; this information is given below under each type of unit's description. After three phases, the units return back to waiting, either dead or alive. After a battle, both units may be alive, dead, or one alive one dead. Dead units, too, are considered waiting as certain unit effects can revive them. The battle between two units can end early if at least one of the battling arena units dies, so not all battles last three phases.

You will implement the following classes for the different units (note that the titles indicate the unit names and class names; for example, any Warrior unit's name is Warrior):

Warrior:
- Maximum health is "level + 2".
- Attack is "level + 1".
- During battle:
  - Phase 1: Do nothing.
  - Phase 2: Attack arena opponent.
  - Phase 3: Attack arena opponent.

Archer:
- Maximum health is "level + 1".
- Attack is "level + 1".
- During battle:
  - Phase 1: Attack arena opponent.
  - Phase 2: Attack an alive random waiting opponent. *Note that some of the actions of certain units will require you to choose a random living waiting opponent or a random dead waiting ally, so it might be useful to have a utility method that does this task in the Unit class: Given an array list of units, choose a random living or dead one, return null if there is no such unit. All phase methods will have access to both units in the arena and both sides' waiting units.*
  - Phase 3: Heal self by 1.

Healer:
- Maximum health is "level + 2".
- Attack is "level".
- During battle:
  - Phase 1: If at maximum health, heal one random waiting ally unit by the amount of "level" (do nothing if there is no waiting alive unit); if not at maximum health, heal self by "level" amount.
  - Phase 2: Attack arena opponent.
  - Phase 3: Revive a random unit among waiting ally dead units (if there is no such dead unit, do nothing).

Rogue:
- Maximum health is "level".
- Attack is "level + 2".
- During battle:
  - Phase 1: Attack arena opponent.
  - Phase 2: Attack arena opponent.
  - Phase 3: Attack an alive random waiting opponent.

Wizard:

- Maximum health is “level” + 2.
- Attack is 1.
- During battle:
    - Phase 1: Do nothing.
    - Phase 2: Do nothing.
    - Phase 3: Attack all opponent’s waiting units.

Bard:

- Maximum health is “level”.
- Attack is “level”.
- During battle:
    - Phase 1: Attack arena opponent.
    - Phase 2: Heal self by 1.
    - Phase 3: Level up one random waiting alive ally unit.

Necromancer:

- Maximum health is “level” + 1.
- Attack is “level”.
- During battle:
    - Phase 1: Attack arena opponent.
    - Phase 2: Revive a random dead ally unit; if there is no dead ally unit, decrease this unit’s current health by one.
    - Phase 3: Decrease the arena opponent’s level by one (it cannot be less than one).

Implement an **Arena** class to keep both sides’ units in two different ArrayLists. These can be **ArrayList<Unit> playerUnits** and **ArrayList<Unit> computerUnits**. Thanks to polymorphism in Java, you do not need to distinguish the subclasses of the Unit as long as they share the common method signatures. The constructor of the **Arena** class should create these array lists and fill them with random units; each type of unit has an equal probability. Each side should have 7 units, and at least one of the units should be a Warrior for both sides. There could be multiple units of the same type. The constructors of the subclasses of the Unit should set the names, starting health, and level of the units.

Include a **void battle(int playerIndex, int computerIndex)** method in your Arena class to send two units of opposite sides to battle. This method should work as follows:

- Remove units at **playerIndex** and **computerIndex** from their respective array lists; a unit in the arena is no longer among the waiting units. You should, of course, keep them in some temporary variable as you need to call their phase methods and return them to the same array list after the battle ends.
- Call the first phase for the battling units. Ensure you pass the parameters considering which side is the enemy and which is the ally for both units. We assume the phases take place simultaneously; of course, you will call the method for one of the sides before the other, but the check for killing the opponent will be made after both units complete their corresponding phase. For example, the player’s unit may kill its arena opponent, and then the computer’s unit may kill the player’s unit, causing both units

to return to their array lists dead. However, if the computer kills our unit without dying itself in one of the phases, then only the player’s unit dies.

- After each phase, check if any of the battling units is dead; if so, the battle ends. Certain unit effects can kill waiting units, but this will not end the battle; the battle continues for three phases while two units are in the arena.
- Call the second phase for the battling units; then check if any battling unit is dead; if so, end the battle.
- Finally, call the third phase for the battling units.
- Before and after each phase, store the dead status of the waiting units on both sides. If any unit’s **isDead** variable is false before the phase and true after the phase (meaning the arena unit killed a waiting one), increase the corresponding arena unit’s level by 1 for each enemy unit killed this way.
- For any of the steps that end the battle, if one arena unit is alive and the other is dead, increase the level of the living unit by 1.
- Add the units in the arena to their respective array lists; you do not need to preserve their original order. The player’s unit should be added to the playerUnits, and the computer’s unit should be added to the computerUnits. Whether the added unit is dead or not is not important; dead units are still kept among the waiting ones in case some other unit revives them.
- Check if the game ended. For this, one of the sides (the player or the computer) should have all of its units dead. If the game ended, set some boolean variable in your Arena class to end the game loop. If not, continue with the newly selected units entering the arena.

Your game loop should continue until the end of the game. At each turn, you will display the units of the player, receive input about which living unit will be sent to the arena (dead units cannot enter the arena), and display information about the battle that takes place; for example, which unit does what at each phase during battle. An example console output is provided in consoleOutput.txt; you may follow the conventions used in that file. The computer should choose one of its living units randomly to send it to the arena. While you display the units to the player, show their index numbers, names, levels, current health, maximum health, and attack points so the player can choose more easily. Also, at the beginning of the game, display the units of both sides so the player can devise a strategy.

Preliminary Submission: You will submit an early version of your solution before the final submission. This version should at least include the following:

- Arena class should be functional, supporting the player to choose a unit to send to the arena at each turn while the game continues.
- Unit class and one of its subclasses should be functional to test it in your game.

You will have time to complete your solution after you submit your preliminary solution. You can consult the TAs and tutors during the lab. Do not forget to make your final submission at the end.

Even if you finish the assignment in the preliminary submission, you should submit for the final submission on Moodle. **Not completing the preliminary submission on time results in 50% reduction of this assignment’s final grade.**