Update Logs:
```
2020-02-23    Initial Version
2020-03-24    Deadline Extension due to Covid-19
2020-04-03    Deadline Extension
```

Due: Tuesday April 7, 2020 11:55PM *(Modified 2020-04-03)*

Learning Objectives:

On the course of implementing this programming project, you will learn some of the basic concepts of object oriented programming and how to apply them to practical, real world programming applications. Specifically, upon accomplishing this project, we expect you to be able to:

1. Differentiate and explain the purposes of:

    - private, public, static, and final class variables.
    - static, void, and returning methods.

2. Implement classes.

3. Instantiate objects from a class and use constructors to initialize objects.

4. Model entities and situations using objects, each of which has certain properties and exhibits certain abilities.

5. Strengthen your overall programming skills.

6. Get a glimpse of needs for abstraction, interfaces, inheritance, and polymorphism.

7. Enjoy coding with Java.

Introduction:

In this project, you will be implementing a program that simulates the gameplay of a turn-based two-team battle game, *Final FiCT*. The skeleton java files are provided. Your task is to understand the gameplay, and implement the methods in the provided skeleton java files which are currently left blank for you to fill in the missing code. A test class (StudentTester) is provided for you to check your outputs and help you to debug your code.

Final FiCT:

Final FiCT is a simple turn-based battle game between two teams: TeamA and TeamB. Similar examples include the battle scenes of Final Fantasy I, II, III, IV, V, VI, VII, VIII, IX, and X, Summoners War, and Fate/Grand Order. However, in this programming project, you will only implement a simple, deterministic mechanism of the game.

Team Alignment:

Players of each team are aligned in two rows: **Front** and **Back**, where the front rows of both teams face each other. The number of players in each row should be specifiable by users. For example, if there are *5* players in each row, then the layout of the arena could be depicted as:
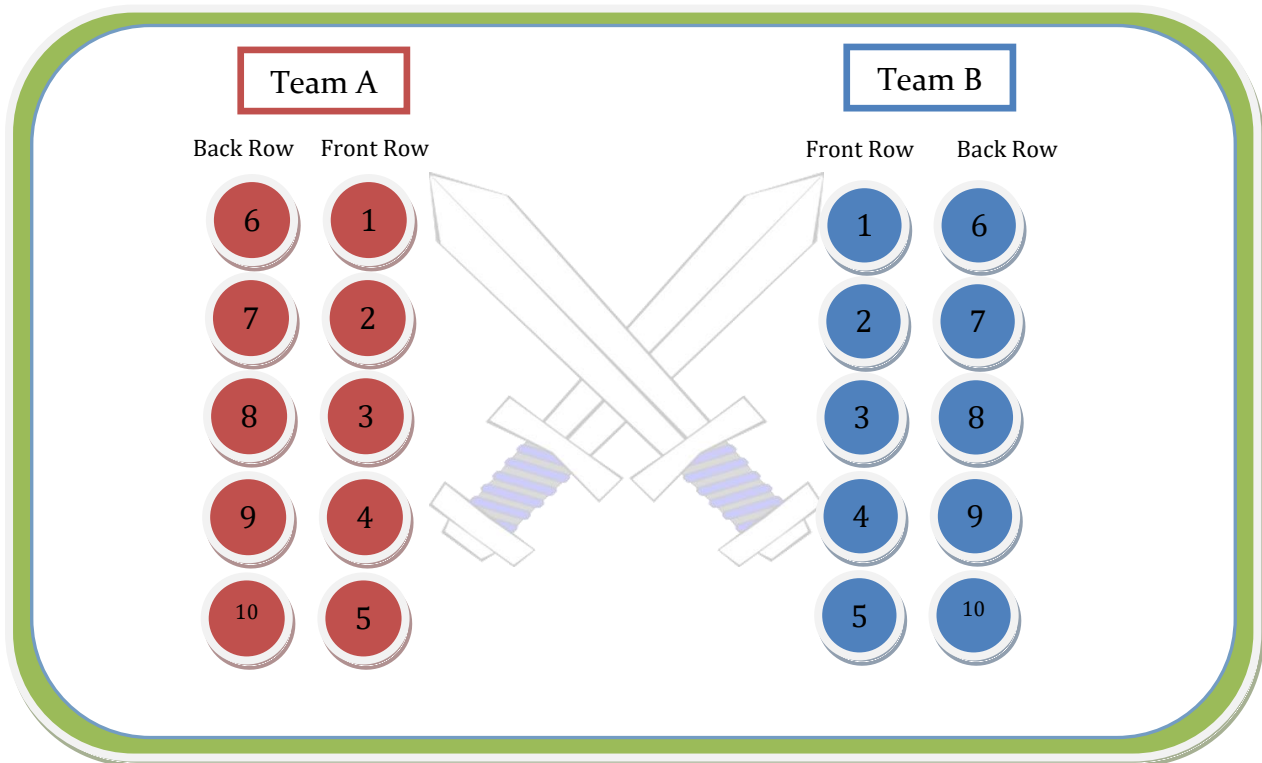
The game must be playable as long as each team has more than one player. In a team, each player has a position number (1 – 10 in the above figure). The position numbers start from top to bottom, and front row to the back row. These position numbers will be used to determine the order of players when attacking and receiving beneficial effects (i.e. healing and reviving, see next section). Typically, the front row players will be eliminated first before moving to the next row. Exceptions to this rule include when a player is taunting, which makes all the players in the opposite team attack just himself.

Tip: Hence, this is probably why positioning tanks (and other high HP players) in the front row is a good idea!

The players:

A player can be one of the following type: `Healer`, `Tank`, `Samurai`, `BlackMage`, `Phoenix`, and `Cherry`. Each player can perform two actions: `attack()` and `useSpecialAbility()`. Each player type has a different set of MAX HP (health points), ATK (attack power), # of Special Turns and special ability. Listed below is the information of each player type:

Type	MAX HP and ATK	Special Ability
Healer	MAX HP: 4790 ATK: 238 # of Special Turns: 4	**Heal** Increase HP of one alive ally with the lowest *percentage* HP by 25% of his MAX HP. If multiple allies have equal lowest percentage HP, heal the first one according to the position order. The HP of the healed ally cannot exceed his MAX HP.
Tank	MAX HP: 5340 ATK: 255 # of Special Turns: 4	**Taunt** Make the players of the opposite team attack (including double-slashing and cursing) himself for one turn[1]. If there are multiple taunting player, the first taunting player according to the position order gets attacked first.
Samurai	MAX HP: 4005 ATK: 368 # of Special Turns: 3	**Double-Slash** Attack the same target twice (even if it is dead after the first attack).
BlackMage	MAX HP: 4175 ATK: 303 # of Special Turns: 4	**Curse** Curse an alive player with the lowest HP in the opposite team in the frontest row. A cursed player does not receive any beneficial effects from the healing abilities for one internal turn *of the curser.* That is the curse status remains until just before it is the *curser*'s turn again (even if the curser is already dead). If a cursed player gets cursed again, the most recent curse takes over the previous curse (hence, the internal clock restarts upon receiving the new curse).

---

[1] A turn is counted by each individual player. For example, once a tank taunts, his taunting effect lasts until just before it is his turn to takeAction() again.

	MAX HP: 4175 ATK: 209 # of Special Turns: 8	**Revive** Revive a dead ally and increase the HP by 30% of his Max HP. If there are multiple dead allies, revive the first one according to the position order. If all the allies are still alive, do nothing. Except for his partial 30% HP, the revived player resumes the battle with a clean slate (i.e. with internal turns and statuses reset). Revived player can resume the battle right away if his turn comes.
Phoenix		
	MAX HP: 3560 ATK: 198 # of Special Turns: 4	**Fortune-Cookies** Lure each of the alive players in the opposite team to eat a sleeping-drug covered fortune cookie, causing them to fall asleep and yield their next turns. That is, the current team gets to take actions for the next turn. While a player is sleeping, its internal clock stops counting (i.e. number of Special Turns freezes). Eating more than one fortune-cookie in one turn yields the same effect as eating just one.
Cherry		

* Pictures are taken from the Internet for illustration purposes only.

In each team, all the positions must be filled. Furthermore, the number of players with the same type must not exceed MAXEACHTYPE (defined in Arena.java).

Gameplay:

The battle is simulated in iterations (or rounds). For each round, each of the alive players (ordered by position numbers) in Team A takes an action. By default, an action is an attack, where the HP of the alive target with lowest HP on the frontmost row of the opposite team is deducted by the attacker's ATK. If a player's HP is reduced to 0, the player is dead, in which case the HP remains 0. Once a player is dead, his curse and taunting statuses (if any) are reset. A dead player cannot perform any action even when his turn comes.

At every "number of Special Turns," the player must perform his special ability. For example, the Samurai's # of Special Turns is 3. Hence, he would perform normal attacks in the first two turns, and double-slash on the third turn.

Once all the players in Team A finish performing actions, Team B then proceeds with the same manner. A round ends when all the alive players finish performing their actions.

The battle is concluded after a round ends AND either one of the following conditions becomes true:

1. All players of a team are dead.

2. The number of simulation rounds exceeds MAXROUNDS (defined in Arena.java).

After the battle terminates, the winning team is the team with the greater number of alive players. If there are equal numbers of alive players, the team with higher sum of HP wins. Otherwise, Team A wins.

Instructions:

In the package you will find the following Java files:

```
Arena.java
Player.java
StudentTester.java
```

Your tasks are to implement the missing code in Arena.java and Player.java, as specified (i.e. see `//INSERT YOUR CODE HERE`).

Arena.java implements the class for the arena that simulates the battle. ***At the end of each round of simulation, you must call `logAfterEachRound()` to log the sum of HP of both teams to battle_log.txt.*** This method is already implemented for you. Each line in the log file should follow the following format:

```
<Round #>    <Sum of HP of Team A>    <Sum of HP of Team B>
```

Examples of log files are included in the provided test cases. We will use these log files to compare against the solutions, so it is your responsibility to make sure that your log files are formatted correctly. You must also make sure that a battle log file (i.e. ***battle_log.txt***) is generated when the program finishes (the log files are needed for grading). ***Do not write your name in the log files.*** We also provide a utility display method `displayArea()`, which prints the current state of the arena, for your debugging purposes.

Player.java implements the blueprint for a player. Some of the basic variables are already provided for you. A player should have an internal turn counter to keep track of when he should perform the special ability. If a player is revived, then the internal turn counter and all the special statuses must be reset.

StudentTester.java is for you to test your code. See Test Cases section.

*DO NOT MODIFY THE JAVA FILE NAMES AND EXISTING METHOD DECLARATIONS as we will test your code against an auto-grader, hence the interfaces must be consistent. However, you are welcome to add your own variables and methods as long as the existing method interfaces and variable names are not altered. You are also not allowed to use third-party Java libraries. Your code should be able to compile and run without having to install external *jar* files.

Test cases:

Two test cases are provided for you to debug your code: `simpleCase()` and `advanceCase()`. See StudentTester.java for the detail. `simpleCase()` simulates a 2x2 game, while `advanceCase()` simulates a 2x5 game. In the testcases folder, you will find the screen output and the battle log files for each case. You should test your code with the `simpleCase()` first and work out the logics by hand.

Coding Style Guideline:

It is important that you follow this coding style guideline strictly, to help us with the grading and for your own benefit when working in groups in future courses. Failing to follow these guidelines may result in a deduction of your project scores.

1. Write your name, student ID, and section (in commented lines) on top of every Java code file that you submit.

2. Comment your code, especially where you believe other people will have trouble understanding, such as purposes of each variable, loop, and chunk of code. If you implement a new method, make sure to put an overview comment at the beginning of each method that explains 1) Objective, 2) Inputs (if any), and 3) Output (if any).

Submission:

It is important that you follow the submission instructions. Failing to do so may result in a deduction and/or delay of your scores.

1. Generate *battle_log.simple.txt*, by renaming `battle_log.txt` generated by `simpleCase()`.

2. Generate `battle_log.advance.txt`, by renaming `battle_log.txt` generated by `advanceCase()`.

3. Put `Arena.java`, `Player.java`, `battle_log.simple.txt`, and `battle_log.advance.txt` in a folder.

4. Rename the folder to P01_<your *Student ID*>.

5. Zip the folder. Make sure the extension of the zip file is .zip (E.x., `P01_62881234.zip`).

6. Submit the zip file on MyCourses before the deadline.

Note: Late submission will suffer a penalty of 20% deduction of the actual scores for each late day. You can keep resubmitting your solutions, but the only latest version will be graded.

Suggestions:

1. Though Final FiCT is a fairly simple game compared to other commercial ones in the market, the small details can overwhelm you during the course of implementation. Our suggestion is to spend some time to understand the requirements from the specs. Then, incrementally implement and test one method/player type at a time, to see if it behaves as expected. ***It is discouraged to implement everything up and test it all at once.***

2. Don't wait until the last weekend to start working on the project. Start early so you have enough time to cope with unforeseen situations. Plus, it can take some time to completely understand the project.

3. Use Java JDK 1.8 or 1.9. Some of the functionalities may not be available in the older versions.

4. Don't use packages yet. Put everything in the default package.

[Optional] Special Challenge (Bonus): Can you beat our team?

Find the team configuration for Team A that can beat the following configuration of Team B, and a great amount of mana (aka. bonus scores) will be bestowed upon you!! We only allocate a pool of 100 bonus points for this project, which will be shared among those who can beat us. For example, if four students can beat us, each of them will earn 25 bonus points. These bonus points will be counted towards the 5% course bonus. So, do not share your answer with your friends! Otherwise, you may get fewer points than you deserve! ☺

Instructors' Team (Team B)	
Front Row	Back Row
Samurai	Healer

Tank	Healer
BlackMage	Tank
Samurai	Phoenix
Samurai	Cherry

If you find a team configuration that can beat our team, submit the following files to "P01 Bonus" on MyCourses.

1. StudentTester.java with implemented `bonusCredit()`

2. Snapshot of the output screen after executing `bonusCredit()`

We will verify your answer on our system, so don't just submit a random configuration. The number of students who win this special battle will be announced on MyCourses. ***The deadline for submitting this bonus is one week prior to project deadline***, so finish early!

Bug Report:

Though not likely, it is possible that our solutions may contain bugs. A bug in this context is not an insect, but error in the solution code that we implemented to generate the test cases. Hence, if you believe that your implementation is correct, yet yields different results, please contact us immediately so proper actions can be taken.

Need help with the project?

If you have questions about the project, please first post them on the forum on MyCourses, so that other students with similar questions can benefit from the discussions. The TAs can also answer some of the questions and help to debug trivial errors. If you still have questions or concerns, please make an appointment with one of the instructors. ***We do not debug your code via email***. If you need help with debugging your code, please come see us.

Academic Integrity (Very Important)

Do not get bored about these warnings yet. But please, please ***do your own work***. Your survival in the subsequent courses and the ability to get desirable jobs (once you graduate) heavily depend on the skills that you harvest in this course. Though students are allowed and encouraged to discuss ideas with others, the actual

solutions must be originated and written by themselves. Collaboration in writing solutions is not allowed, as it would be unfair to other students. Students who know how to obtain the solutions are encouraged to help others by guiding them and teaching them the core material needed to complete the project, rather than giving away the solutions. **You can't keep helping your friends forever, so you would do them a favor by allowing them to be better problem solvers and life-long learners.** Your code will be compared with other students' (both current and previous course takers) and online sources using state-of-the-art source-code similarity detection algorithms which have been proven to be quite accurate. If you are caught cheating, serious actions will be taken, and heavy penalties will be bestowed on all involved parties, including inappropriate help givers, receivers, and middlemen.

Mastering coding takes practice and effort.