Computer Laboratory 8

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Essential information

- This assignment is due Tuesday Nov 7th at noon and will be turned in on gradescope
- The assignment is intended to be done in pairs, although you're allowed to work on your own if you want.
- You can start the assignment before your lab period, although I recommend not getting too far – the lab TAs often have useful hints, and I don't want you wasting too much time struggling with something that the TAs might announce in-class.
- For more rules see the lab rules document on canvas.
- Like previous java labs – this lab will have a large function count – many of which are pretty short and easy to program correctly. Do not let the length scare you, but also make sure you're not spending too much time stuck on the "easy" functions!

2 Introduction

In this lab we will be making our own version of the popular "Pokemon" franchise of video-games: the CodeMonsters! CodeMonsters are each based on various programming languages. With powerful moves like "recursion", "iteration", and "strongly typed" the fun won't stop!

In all seriousness though – video games often have a myriad of complicated data representation tasks that are often uniquely well suited to object oriented programming such as seen in the Java programming language. Doing this java lab will give us a lot of practice building java objects, a chance to explore how objects can interact with each other in various ways, as well as an introduction to basic inheritance and polymorphism (these are advanced topics, but ones that this problem illustrates well, so we figured we would preview it here!) Like previous labs, the algorithms you're going to implement will often be pretty simple, instead the challenge will be in understanding the java concepts and syntax for object oriented programming. Make sure you're focusing on these aspects and ensuring you fully understand why each piece of code must be how it is.

3 Software environment setup

This lab will be done using a java programming environment. We recommend the IntelliJ IDE running on your own personal computer. See lab 6 for instructions about how to set this up.

A few reminders:

- Source code goes in the `src` folder in java. Java is VERY particular about where files go, and what they are called.
- Lab06 has syntax guides if you still find translating to java difficult.
- Do not have any folders inside src – these are called packages and will make your code fail the autograder.
- Download *every* provided file and move them all to the src folder before starting.
- Code style elements like comments and javadocs are **important** and **easier to do as you go**. Don't write hard-to-read code and think "I can fix it later".

4 Files

This lab will involve the following **provided** files.

- `BattleTester.java` – a tester file for one of the required classes.
- `CodeMonsterTester.java` – a tester file for one of the required classes.
- `HealingSkill.java` – an example file which you can reference to see how two of the other classes you will have to program will be written.
- `MoreSkillsTester.java` – a tester file for two of the required classes.
- `SkillTester.java` – a tester file for one of the required classes.

This lab also involves the following files **you will create**:

- `CodeMonster.java` – a class that represents one CodeMonster
- `Skill.java` – a class that represents one of the CodeMonster's "moves" (I.E. one thing they can do on their turn)
- `Battle.java` – a class that has static functions to run a CodeMonster battle!
- `VampiricSkill.java` – a specialized skill that hurts and heals!
- `FastSkill.java` – a specialized skill that takes no time at all!

5 Instructions

Before beginning you should:

1. Setup an IntelliJ package
2. Download the provided files and place them in the src folder.

3. Create empty versions of the 5 required java files, and add empty versions of all required functions. **DO NOT SKIP THIS STEP** – the Skill and CodeMonster files depend on each other – so it will be a lot easier to program if you get both classes in-place with empty functions before programming any of it.
4. Ensure that your name, and your partner's name, is at the top of each file
5. Skim following sections and make sure you understand the code you are about to write – and how each major part of it works.

Once you're done with these steps, it's reasonable to start programming. All of these steps *before* coding can reasonably be discussed with others (even those outside your group), but please solve the actual programming-parts of this problem collaborating only with your lab partner.

6 Overview of required functions

To help you with creating empty files, the following is a simple listing of all required functions without further explanation. These are all explained in later sections

- `Skill`
  - `public Skill(String name, int strength, int usageLimit)`
  - `public String getName()`
  - `public void refresh()`
  - `public int getStrength()`
  - `public int getUsageLimit()`
  - `public int getUsageLeft()`
  - `public void useSkill(CodeMonster me, CodeMonster foe)`
  - `public void applyChanges(CodeMonster me, CodeMonster foe)`
  - `public String toString()`
- `VampiricSkill`
  - `public VampiricSkill(String name, int strength, int usageLimit)`
  - `public void applyChanges(CodeMonster me, CodeMonster foe)`
- `FastSkill`
  - `public FastSkill(String name, int strength, int usageLimit)`
  - `public void applyChanges(CodeMonster me, CodeMonster foe)`
- `CodeMonster`
  - `public CodeMonster(int maxHp, double speedScore, String name, Skill[] moves)`
  - `public void prepForBattle()`
  - `public Skill takeTurn()`
  - `public boolean isAlive()`
  - `public void adjustHealth(int amount)`

  - `public void setNextTurnTime(double nextTurnTime)`
  - `public double getNextTurnTime()`
  - `public int getHp()`
  - `public int getMaxHp()`
  - `public Skill[] getMoves()`
  - `public String getName()`
  - `public double getSpeedScore()`
  - `public String toString()`

- `Battle`
  - `public static void doOneTurn(CodeMonster one, CodeMonster two)`
  - `public static CodeMonster battle(CodeMonster one, CodeMonster two)`

7 Skill

The easiest place to start programming is likely the “Skill” class. The skill class represents one action a CodeMonster can take in battle.

The Skill class you make should have 4 private attributes:

- a name (String)
- a strength score (int) which represents the “power” of the skill – in this case how much damage it will do.
- and a pair of ints to represent the how many times the skill can be used in one battle (one to track the max, and one to track how many uses are left this battle)

You should add these attributes to your code, as well as the following getter functions (which should be easy 1-liners that return the current value of their respective attribute)

- `public String getName()`
- `public int getStrength()`
- `public int getUsageLimit()`
- `public int getUsageLeft()`

Now would also be a good time to implement the following relatively straightforward functions:

- `public Skill(String name, int strength, int usageLimit)` This constructor should assign the private name, strength, and usage limits from it’s input parameters. Additionally, set up variables so that getUsageLeft would return the usage Limit (I.E. a newly created skill has all it’s uses left.)
- `public void refresh()` This should reset private variables so that getUsageLeft will be the usageLimit. This function will be called at the beginning of a battle to reset the use-tracking variables for a new battle!

- `public String toString()` This should return a string formed of the name, a space, the number of remaining usages, a "/" and the usage limit. For examples of this format see test code.

The remaining two functions require a bit more explanation:

7.1 applyChanges

The function `public void applyChanges(CodeMonster me, CodeMonster foe)` is in charge of actually updating the state of the CodeMonsters to reflect the change made by using this skill. It may seem strange to separate this into it's own function, but it will end up being the right decision later, when we introduce other versions of Skills.

This class (Skill) represents a basic attack, so "applying the changes" of having "me" attack "foe" would mean dealing damage to foe (or more directly, reducing it's health). You will need to use the adjustHealth method on class CodeMonster to do this. Note, the amount to reduce the health by is the current power score, so a Skill with power 10 should do 10 damage (**reduce** the foe's health by 10). You may want to at least re-skim the adjustHealth function to make sure you know how it treats parameters, you could also go work on CodeMonster and come back to this function and the next one later if you wish.

7.2 useSkill

The function `public void useSkill(CodeMonster me, CodeMonster foe)` is the main entry point to using a skill that the battle code will use. In our code, however, this function has only one responsibility – ensuring usage limit for a skill is enforced. This function should ensure that there are usages left for the current skill. If there are reduce the remaining usages by one and then call applyChanges to actually effect the battle. If the move does not have any uses left this battle this function should do nothing.

8 CodeMonster

The CodeMonster class represents a single code monster in our game. This class has more attributes and functions than skill, and these form groups of related behavior.

You will ultimately need 7 attributes:

- the name (String)
- a pair of ints to track the current and maximum HP (health points)
- the Skill array (named moves) of moves the CodeMonster can use.
- an int to track which moves the monster used recently – more in this later. I'll refer to this as "moveIndex" but you may find other names useful for you.
- a pair of double values to track how fast the monster is, and when it will take it's next turn (speedScore and nextTurnTime respectively) CodeMonster battles **do not take turns** but instead use a time-based system so that faster monsters can take

more actions than slower monsters. For reference, smaller speed scores indicate faster monsters.

You should begin by adding these attributes, as well as the following getters (each of which should simply return the value of their related variable) Each should be an easy 1-line function.

- `public double getNextTurnTime()`
- `public int getHp()`
- `public int getMaxHp()`
- `public Skill[] getMoves()`
- `public String getName()`
- `public double getSpeedScore()`

The following should also be straightforward to write:

- `public CodeMonster(int maxHp, double speedScore, String name, Skill[] moves)` This constructor should update the maxHP, speedScore, name, and moves array from it's parameters. The current hp should be set to the maxHP. The nextTurnTime should be set to the speed score.
- `public boolean isAlive()` This function should return true if the current HP is greater than 0, false otherwise.
- `public void setNextTurnTime(double nextTurnTime)` This function should update the private nextTurnTime attribute based on it's input parameter value.
- `public String toString()` This function should return a string that is the CodeMonster's name, followed by a space, then their current hp, then a "/" then their max hp. Examples of this format can be seen in the tester code.

The remaining functions will require a bit more discussion.

8.1 adjustHealth

The `public void adjustHealth(int amount)` method should *adjust* the current HP with positive amounts increasing the current HP (representing healing) and negative amounts decreasing the HP (representing damage). This function is in charge of enforcing two limits: the HP should never go over the maximum, and the HP should never go below 0 (although it can be reduced to 0). If the hp would ever be reduced below 0 in this function simply set the hp to 0. If the hp would ever be increased above the max in this function, simply set the hp to it's max.

8.2 takeTurn

The `public Skill takeTurn()` function will be called by the battle code when it's this codeMonster's turn to use a skill. This function has two seperate, but related tasks:

- update the nextTurnTime – this is done by adding the codeMonster's speed to the next turn time. In this way faster codeMonster's (smaller speed scores) will act more often (as their next turn time will not increase as much after acting).
- choosing which skill to use. This is indicated by returning the Skill the CodeMonster wishes to use. We will use a simple strategy here – CodeMonsters will use the skills in their moves array in the order given in the array – looping over the array after each skill has been used.
  So if a CodeMonster has 3 moves (for example, let's call them A, B, and C, in that order) then the first time takeTurn is called, it should return A. The second time takeTurn is called B should be returned, the third time takeTurn is called C should be returned, and then the fourth time A should be returned, the fifth time B should be returned and so-forth. (Note – no consideration should be made for if the skill can be used more times. Even if a skill cannot be used again, it should be returned)
  You will need to use the "moveIndex" variable – or some other private variable of your own devising to accomplish this.

8.3 prepForBattle

The method `public void prepForBattle()` is called before a new battle. This should reset private variables for a new battle as follows:

- the current hp should be reset to max
- the next turn time should be set to the codeMonster's speed score.
- the "moveIndex" (or whatever you named that variable) should be reset such that the next call to takeTurn will return the first skill in the array.
- every skill in the moves array should be refreshed, using the skill refresh method (so that each move's use-count is reset)

9 Battle

This class should only have static methods.

9.1 doOneTurn

The `public static void doOneTurn(CodeMonster one, CodeMonster two)` function is in charge of handling exactly one turn of battle. Normally this would be a private function, but we are making it an explicit part of the problem design to help us test your code for correctness, and help you structure the battle logic. This function should first figure out which codeMonster (one or two) is taking this turn. Note – only one CodeMonster should act each time this function is called – codeMonsters do not take equal numbers of turns – faster codeMonsters act more frequently than slower ones. The codeMonster with a smaller nextTurnTime should go, if the codeMonsters are tied for nextTurnTime then codeMonster one should go.

After deciding which codeMonster is going, it's takeTurn method should be called to decide which skill is used, and the skill should be used. Finally, a message should be printed – see the tester code for the format of this message. (As a hint – the message is deliberately designed to large be made up of the toString outputs from the Skill and CodeMonster class – make sure you're using those methods!)

9.2 battle

The `public static CodeMonster battle(CodeMonster one, CodeMonster two)` function will carry out an entire battle.

- start by preparing each codeMonster for battle
- Then print a message indicating the match up (see tester code for exact message format.)
- then, one turn should be taken at a time until one codeMonster is no longer alive.
- Once one codeMonster is not alive, the living codeMonster is declared the winner.
- A message should print to note who won (again – see tester code for exact message format.) and the winner should be returned to indicate the victory.

10 More Skills

One interesting opportunity we have here is to use the advanced object oriented features *inheritance* and *polymorphism* to allow codeMonsters to have more complicated moves. Since this will be a bit ahead of discussing these features in-class, we have given you *an entire working example!*

10.1 HealingSkill

The healingSkill class is a child-class of the Skill class. While it only *declares* a constructor and an applyChanges method, it has all methods from Skill. Please note the following syntax from this class:

- `public class HealingSkill extends Skill` the special keyword extends sets up HealingSkill as a child class of Skill. This will be required for the two classes you write.
- (in the constructor) `super(name, strength, usageLimit);` this code invokes the Skill constructor – We will discuss this in class, but this is something each child-class constructor will need to do.
- (in applyChanges) `getStrength()` Note that we have to use the getter here – we cannot use the `strength` attribute directly because it is *PRIVATE* – and access to private variables is not inherited. Your code will need this too!
- Notice further that no new attributes were added to this class. They are not needed, as we inherit attributes for name, strength, and usage tracking from the Skill class.

10.2 FastSkill

The fastSkill represents skills that can be used in no time at all! When applied, this skill should damage the foe based on the skill's power. and reduce the "me" codeMonster's nextTurnTime by that codeMonsters speed score. (in essence refunding the time that was marked for this turn, and ensuring the codeMonster goes again!)

10.3 VampiricSkill

The vampiric skill represents a skill that hurts the foe and heals the current codeMonster at the same time. When applied, this skill should damage the foe based on the skill's power, and also heal the "me" codeMonster by the skill's power.

10.4 Personal testing

To keep tests simple for students who may struggle on this last part we have not added any tests that formally show off how these skills can be used (as such tests might keep you from compiling your code...). As such, we recommend you spare a few minutes to see this for yourself. The following is strongly recommended, but will not be graded. None the less you **should do this** to help you better understand what we can do with polymorphism and why we would build our software this way!

Create a main function on some class and create two codeMonsters based on two programming languages you know. Give them each a few different skills. As you do this, try to give them a healing skill, vampiric skill, and fastSkill – this should require no special syntax, simply make the objects and assign them into a Skill array. You should be able to see the effects of each of these in practice by "watching" your do codeMonsters battle!

If you've done everything right, you'll see that you can simply add the new skills to a Skill array as if they were normal skills! That's the power of polymorphism.

11 Java Autograder Notes

The java autograder is much fussier than the python autograder. This is mostly due to Java's nature as a compiled language. While we could often *partially* test python files that had errors in some functions – for your code to be testable your code must compile, have all the correct function definitions, and all the correct names. Even once you get the tests *running* you should also be aware that the autograder for java does not give as clear feedback as the python autograder when things go wrong. As such **you are expected to be testing you code on your own**. Do not use the autograder as a debugging tool.

Due to some strange things in the autograder, we have a few specific requirements for your code:

1. The code you submit must be valid and compileable java code. (We cannot test code with syntax errors. If we can't test it, we can't give you autograder credit for it)

2. The code you submit must have a student package statement in it. (We have to do some weird things with package structures to get the code to compile correctly.)
3. The code you submit must match the provided function signature (name, parameter types, and modifiers) EXACTLY. Any mis-match, no matter how minor, may cause the test code to not compile, which would prevent testing.

12 Submission

For this lab you need to turn in:

- `VampiricSkill.java`
- `Skill.java`
- `FastSkill.java`
- `CodeMonster.java`
- `Battle.java`