Homework 04 – Super Java Pets

Authors: Lucas, Jack, Garrett, Chelsea
Topics: Polymorphism, dynamic binding, casting, Interfaces, comparable

Problem Description

Please make sure to read all parts of the document carefully.
After spending several hours playing other pet-related auto battler games on Steam, you realized you had become way too good at it, so you decide to make your own! Let's start our game development career with this homework assignment.

Solution Description

For this assignment, you will create Pet.java, Turtle.java, Hippo.java, Skunk.java, and PetBattlefield.java. Pay special attention to the hierarchical relationship between the classes and how to reuse as much code as possible.

@CS131

Pet.java

This will be an abstract class representing a Pet. Pets should be able to be compared based on a combination of their stats, health, and attack, properly following the contract set by the Comparable interface using generics.

Variables

- `health`
  - An `int` representing the health points for this Pet.
  - **Note:** A Pet may reach zero or negative health by battling others but cannot have zero or negative health to begin with. If this condition is met, the Pet is considered to be fainted.
  - In the case that we try to instantiate a Pet with a **nonpositive** health, we default to a health of 1.
- `attack`
  - An `int` representing the attack points for this Pet.
  - In the case that we try to instantiate a Pet with a **negative** attack, we default to a value of 0.

Constructor(s):

- A constructor that takes `health` and `attack`, setting those values according to the above specifications.

Methods:

- `hasFainted`
  - Return a `boolean` representing whether this Pet has fainted.
- `getAttacked`
  - Takes in an `int` representing the amount of damage this Pet is being attacked for.
  - Make sure that the Pet's health is decreased by the damage it took.
- `attackPet`
  - Takes in another `Pet` that will be attacked by this Pet.
  - **Note:** Assume the passed in `Pet` will not be null.
  - Calls `getAttacked` on the argument with the damage done (its attack) by this Pet.
- `toString`
  - This method should properly override the parent class's method of the same name.
    - Note: Specifying @Override before the method header allows you to skip the Javadocs for that method and tells the compiler to double check that the method has been properly overridden.
  - A string representation of a Pet looks like this:

    `"<attack>/<health>"` (without the angle brackets)
- `compareTo`
  - This method should properly override the Comparable interface's method of the same name.
  - A Pet is considered (less than/equal to/greater than) another if the sum of its health and attack are (less than/equal to/greater than) the sum of the other's health and attack.
  - If the input is null, return as if this Pet is greater than it (any positive integer).
- Getters and setters as necessary.

Turtle.java

This will be a class representing a Turtle, a subclass of Pet. A Turtle can start with melon armor which prevents up to 20 damage the **first time** it is attacked.

Variables:

- `melonArmor`
  - A `boolean` representing whether this Turtle possesses melon armor, the ability to block 20 damage from an incoming attack once.

Constructor(s):

- A constructor that takes `health`, `attack`, and `melonArmor`. (What keyword allows us to continue to follow the specifications of the Pet class by reusing code?)
- A constructor that takes no arguments that sets `health` to `4`, `attack` to `2`, and `melonArmor` to true.

Methods:

- `getAttacked`

CS131

    - This method should properly override the parent class's method of the same name.
    - If the Turtle possesses melon armor, prevent up to 20 damage from the **first** attack it receives.
- `toString`
    - This method should properly override the parent class's method of the same name.
    - A String representation of a Turtle looks like this:

      `"Turtle:<attack>/<health>/<melonArmor>"` (without the angle brackets)

    - Think about which keyword allows us to minimize the code written in this method.
- Getters and setters as necessary.

Hippo.java

This will be a class representing a Hippo, a subclass of Pet. A Hippo can get stronger, or buffed, each time it makes an enemy Pet faint.

Variables:

- `buff`
    - An `int` representing the buff, or increase to this Hippo's health and attack, it will receive if it successfully knocks out another Pet.
    - This value should not be negative. In the case that we attempt to instantiate a Hippo with a negative buff, you should instead make it 0.

Constructor(s):

- A constructor that takes `health`, `attack`, and `buff`. (What keyword allows us to continue to follow the specifications of the Pet class by reusing code?)
- A constructor that takes no arguments that sets `health` to 7, `attack` to 4, and `buff` to 2.

Methods:

- `getBuffed`
    - This method should increase the Hippo's health and attack by `buff`.
- `attackPet`
    - This method should properly override the parent class's method of the same name.
    - Calls `getAttacked` on the argument with the proper damage done by this Hippo.
    - If the attack successfully causes the attacked Pet to faint, this Hippo should get buffed.
- `toString`
    - This method should properly override the parent class's method of the same name.
    - A String representation of a Hippo looks like this:

      `"Hippo:<attack>/<health>/<buff>"` (without the angle brackets)

    - Think about which keyword allows us to minimize the code written in this method.
- Getters and setters as necessary.

©CSI331

Skunk.java

This will be a class representing a Skunk, a subclass of Pet. A Skunk can make other Pets weaker before attacking them but only a set number of times.

Variables:

- `numSpray`
  - An `int` representing the number of times the Skunk can spray.
  - This value should not be negative. In the case that we attempt to instantiate a Skunk with a negative numSpray, you should instead make it 0.

Constructor(s):

- A constructor that takes `health`, `attack`, and `numSpray`. (What keyword allows us to continue to follow the specifications of the Pet class by reusing code?)
- A constructor that takes no arguments that sets `health` to 5, `attack` to 3, and `numSpray` to 1.

Methods:

- `sprayPet`
  - Takes in another `Pet` that will be sprayed by this Skunk.
  - Reduces the attack and health of the Pet sprayed by a third of its original stats
  - Reduces the number of times this Skunk can spray in the future by one.
- `attackPet`
  - This method should properly override the parent class's method of the same name.
  - If the Skunk can still spray, the Skunk should spray the other Pet once before dealing its damage to it.
  - Calls `getAttacked` on the argument with the proper damage done by this Skunk.
- `toString`
  - This method should properly override the parent class's method of the same name.
  - A String representation of a Skunk looks like this:

    `"Skunk:<attack>/<health>/<numSpray>"` (without the brackets)

  - Think about which keyword allows us to minimize the code written in this method.
- Getters and setters as necessary.

PetBattlefield.java

This will be a class representing where Pet battles will occur.

Variables:

- `firstTeam`
  - An array of Pets with a maximum capacity of 5

@CS131

    - **Note:** Null entries are allowed
- `secondTeam`
    - An array of Pets with a maximum capacity of 5
    - **Note:** Null entries are allowed

Constructor(s):

- A constructor that takes `firstTeam` and `secondTeam` and sets them appropriately. If either array has a capacity of more than 5, replace both with empty arrays of size 5.

Methods:

- `toString`
    - This method should properly override the parent class's method of the same name.
    - A String representation of a PetBattlefield looks like this:

      `"First Team: <Pet0, Pet1, Pet2, Pet3, Pet4> vs Second Team: <Pet0, Pet1, Pet2 Pet3, Pet4>"` (without the angle brackets).

    - Use the `toString` of each Pet to visualize each team!
    - If any Pet is null, print `"Empty"` for that index instead.
- `compareTeams`
    - Compare each Pet in the first team with its corresponding Pet in the second team to decide which team is more likely to win solely based on their stats.
    - The team that has the most Pets that are considered greater than the other wins.
    - Once you decide, print

      `"The <first/second> team will probably win."` (without the angle brackets).

    - If there is a tie, print

      `"It is an even match."`

- `battle`
    - Start a battle between the two teams in the PetBattlefield until at least one of the teams has only fainted pets.
    - The battles should occur starting with the Pet at the first index of each team.
    - The Pet that does not faint goes on to fight the Pet at the second index of the other team and so on. If neither Pets faint with just one attack each, they should continue to attack each other until one does. **Note:** Once a Pet faints, it should be made **null** in its corresponding team.
    - Note that whenever Pet A attacks Pet B, Pet A gets simultaneously attacked by Pet B. **However**, you should give priority to the abilities (melon armor, getting buffed, or skunk spray) of the first team to trigger before the second team's abilities.
        - Allow for both teams to attack once before checking if either Pet faints.
    - To indicate the winning team, print

      `"The <first/second> team won!"` (without the angle brackets).

    - If there is a tie, print

CS131

```
"Both teams fainted."
```

- Getters and setters as necessary.
- `main`
  - Use this to test the functionality of your auto battler!
  - Compare at least two teams.
  - Create at least one battle with at least one of each Pet. Include some empty (aka. null) Pets in both teams.
  - Use the toString methods of each class to your advantage to see what your code is doing!

Checkstyle

You must run checkstyle on your submission (To learn more about Checkstyle, check out cs1331-style-guide.pdf under CheckStyle Resources in the Modules section of Canvas.) **The Checkstyle cap for this assignment is 25 points.** This means there is a maximum point deduction of 25. If you don't have Checkstyle yet, download it from Canvas -> Modules -> CheckStyle Resources -> checkstyle-8.28.jar. Place it in the same folder as the files you want to run Checkstyle on. Run checkstyle on your code like so:

```
$ java -jar checkstyle-8.28.jar yourFileName.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be the points we would take off (limited by the checkstyle cap mentioned in the Rubric section). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

Additionally, you must Javadoc your code.

Run the following to only check your Javadocs:

```
$ java -jar checkstyle-8.28.jar -j yourFileName.java
```

Run the following to check both Javadocs and Checkstyle:

```
$ java -jar checkstyle-8.28.jar -a yourFileName.java
```

For additional help with Checkstyle see the CS 1331 Style Guide.

Turn-In Procedure

Submission

To submit, upload the files listed below to the corresponding assignment on Gradescope:

- `Pet.java`
- `Turtle.java`
- `Hippo.java`
- `Skunk.java`

©CS1331

- `PetBattleField.java`

Make sure you see the message stating the assignment was submitted successfully. From this point, Gradescope will run a basic autograder on your submission as discussed in the next section. **Any autograder test are provided as a courtesy to help "sanity check" your work and you may not see all the test cases used to grade your work.** You are responsible for thoroughly testing your submission on your own to ensure you have fulfilled the requirements of this assignment. If you have questions about the requirements given, reach out to a TA or Professor via Piazza for clarification.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the homework. We will only grade your latest submission. **Be sure to submit every file each time you resubmit**.

Gradescope Autograder

If an autograder is enabled for this assignment, you may be able to see the results of a few basic test cases on your code. Typically, tests will correspond to a rubric item, and the score returned represents the performance of your code on those rubric items only. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue.

The Gradescope tests serve two main purposes:

- Prevent upload mistakes (e.g. non-compiling code)
- Provide basic formatting and usage validation

In other words, the test cases on Gradescope are by no means comprehensive. Be sure to thoroughly test your code by considering edge cases and writing your own test files. You also should avoid using Gradescope to compile, run, or Checkstyle your code; you can do that locally on your machine.

Other portions of your assignment can also be graded by a TA once the submission deadline has passed, so the output on Gradescope may not necessarily reflect your grade for the assignment.

Allowed Imports

To prevent trivialization of the assignment, you are not allowed to import any classes or packages.

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our auto grader. For that reason, do not use any of the following in your final submission:

- `var` (the reserved keyword)
- `System.exit`
- `System.arraycopy`

Collaboration

Only discussion of the Homework (HW) at a conceptual high level is allowed. You can discuss course concepts and HW assignments broadly, that is, at a conceptual level to increase your understanding. If you find yourself dropping to a level where specific Java code is being discussed, that is going too far.

@CS131

Those discussions should be reserved for the instructor and TAs. To be clear, you should never exchange code related to an assignment with anyone other than the instructor and TAs.

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Do not submit `.class` files
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- **Check on Ed Discussion for a note containing all official clarifications and sample outputs**

It is expected that everyone will follow the Student-Faculty Expectations document, and the Student Code of Conduct. The professor expects a **positive, respectful, and engaged academic environment** inside the classroom, outside the classroom, in all electronic communications, on all file submissions, and on any document submitted throughout the duration of the course. No inappropriate language is to be used, and any assignment, deemed by the professor, to contain inappropriate, offensive language or threats will get a zero. You are to use professionalism in your work. Violations of this conduct policy will be turned over to the Office of Student Integrity for misconduct.

©CS1331