Homework 02 – Jurassic Park Rancher

Authors: Brandon, Ruchi, Daniel, Sang Yoon
Topics: Javadocs, Inheritance, Constructor Chaining, Copy Constructor, Static Variables, Wrapper Classes, Good Class Design

Problem Description

You have just been hired to be a rancher at the new attraction, Jurassic Park! There's much to do with maintaining all the enclosures for all these precious, harmless reptiles. Fortunately, you've decided to apply your object-oriented skills to build a management system for the park. Surely none of them will escape this time!

Solution Description

You'll be creating the 5 classes in this assignment: Dinosaur, Pterodactyl, Velociraptor, Pack and JurassicPark. Each of these has constructors, methods, and variables that are described below. A lot of the code will be reused through inheritance, so make sure you pay attention to the hints given below!

Notes:

1. All variables should be inaccessible from other classes and require an instance to be accessed through, unless specified otherwise.
2. All methods should be accessible from everywhere and must require an instance to be accessed through, unless specified otherwise.
3. Use constructor chaining and reuse methods whenever possible! Ensure that your constructor chaining helps you maximize code reuse!
4. Reuse code when possible and helper methods are optional.
5. Make sure to add Javadoc comments to your methods and classes!

Dinosaur.java

This file defines a Dinosaur object.

Variables:
The `Dinosaur` class must have these variables:

- `name` – the name of the dinosaur. This should be a `String`. It should be visible to subclasses of `Dinosaur`. An invalid value should default to "`Dino`". The value is invalid if name is an empty String, blank String, or null. This value *cannot* be changed once set and should be visible to any descendant classes. Think about what modifiers fit best for this!
- `height`– a `double` representing the height of the dinosaur in feet.
- `width`– a `double` representing the width of the dinosaur in feet.
- `weight`–a `double` representing the weight in pounds.
- `totalEnclosures` – an `int` representing the total number of dinosaurs that have an enclosure at the park. Think about what reserved word you need to use for this variable. It should be visible to subclasses of `Dinosaur`.

Constructors:

- A constructor that takes in the `name`, `height`, `width`, and `weight`.
- A constructor that takes in no parameters. `name` should be set to "`Dino`". `height` should be set to 15, `width` should be 20 and `weight` should be set to 1000.
- A copy constructor that deep copies all necessary instance variables of the old object to the new object.

Methods:

- `enclosureSize()`
  - Returns a `double` representing the area of the enclosure where the dinosaur is located.
  - To calculate, multiply 10 times the `width`, followed by the `height`.
- `calculateFood()`
  - Returns a `double` representing the amount of food in pounds needed to prepare for the dinosaur
  - To calculate, multiply `weight` times the `width`, followed by the `height`.
- `toString()`
  - Returns the `String` representation of the object.
  - Should return a `String` in the following format:
    `"<name> requires a <enclosureSize> square foot enclosure and <calculateFood> pounds of food."`
- `buildEnclosure()`
  - Each dinosaur has a given area for its enclosure and a set of amount of food. If the calculations from `calculateFood()` and `enclosureSize()` exceeds that limit, then we cannot build the enclosure.
  - If `enclosureSize()` exceeds 6000 or `calculateFood()` exceeds 80000, return a a String of `toString()` with the following concatenated at the end of the String:
    `" <name> is too expensive for the park!"`
    - **Note: There is a single space before `<name>`.**
  - Otherwise, return a String of `toString()` with the following concatenated at the end of the String:
    `" <name> has been added to the park!"`
    - **Note: There is a single space before the `<name>`.**
  - Make sure to update all relevant variables in this method (i.e. increment the total number of dinosaurs in an enclosure when an enclosure is successfully built.)
- `numPlatesRequired()`
  - Each dinosaur has a sign in front of its temporary cage having the basic information of that dinosaur. However, since `height`, `width`, and `weight` are values that can change, each of their digits are written as exchangeable plates.
  - Returns an `int` representing the number of minimum exchangeable plates that is needed to describe `height`, `width`, and `weight` of the dinosaur.
    - **Note: The number of minimum exchangeable plates for each value is 3: one plate for the decimal point, at least one plate for the first decimal digit and one for the first non-decimal digit**
    - **Hint**: We know ways to easily get the length of a `String`. We also know ways to convert a primitive data type to a wrapper class. Is there method in the wrapper class that would give us a String representation of it?
- Appropriate getters and setters for each of the instance and static variables.

Pterodactyl.java

This file defines a Pterodactyl object, which is a subclass of Dinosaur. Since they can fly, their enclosures need to account for the altitude they fly up to.

Variables:

In addition to the variables it inherits from its superclass, `Pterodactyl` should also have the following variables:

- `flightCeiling` – a `double` representing the altitude the Pterodactyl can fly up to in the range [10, 100] in feet. If an invalid value is entered, the variable should default to 50.

Constructors:

- A constructor that takes in the `name`, `height`, `width`, `weight`, and `flightCeiling`.
- A constructor that takes in the `name`, `width`, and defaults `flightCeiling` to 50, and all other variables to their default in `Dinosaur`.
- A constructor that takes in `name` and defaults `width` to 12. All other variables should end up initialized to their corresponding defaults in `Dinosaur` and `Pterodactyl`.
- A copy constructor that deep copies all instance variables of the old object to the new object.
- **Hint:** Make sure to utilize constructor chaining whenever possible, and don't forget you can constructor chain to a parent class from a child class!

Methods:

- `enclosureSize()`
    - Returns a `double` representing the area of the enclosure where the dinosaur is located.
    - To calculate, multiply four times the `width`, followed by the `height`. Then add `flightCeiling` to this value.
    - Override the `enclosureSize()` method from `Dinosaur`
- `toString()`
    - Returns the String representation of the object.
    - Should return a String in the following format:
      `"<name> can fly <flightCeiling> feet into the air! <name> requires a <enclosureSize> square foot enclosure and <calculateFood> pounds of food.`
- Getters and setters for each of the instance variables.

Pack.java

This object will store details about the pack a dinosaur may belong to. Dinosaurs that belong to a pack will require more food and larger enclosures.

Variables:

- `size` – an `int` that represents the size of the pack in number. If an invalid value is entered, then the default value should be 4. An invalid value would be a negative number.
- `packName` – a `String` that represents the name for this group of dinosaurs on the sign of the enclosure. An invalid value should default to "`The Power Pack`". The value is invalid if `packName` is attempted to be set to an empty String, blank String, or null.
- **Note: Both fields should be immutable, making the instances of this class effectively immutable. This means we do <u>not</u> need a copy constructor for this class.**

Constructors:
- A constructor that takes in `size` and `packName`

Methods:
- `toString()`
    - Returns a `String` representation of the object in the given format: `"<packName> is a family of dinosaurs of size <size>!"`
- Getters and setters when necessary

Velociraptor.java

This file defines a Velociraptor object and should be a subclass of Dinosaur. The velociraptor runs a lot, which means they eat a greater amount of food compared to the regular dinosaur. Some velociraptors may belong to a pack in an enclosure or may hunt alone.

Variables:
In addition to the variables it inherits from its superclass, `Velociraptor` should also have the following variables:
- `speed` – an `int` representing the speed of the velociraptor in miles per hour. An invalid value should be set to 30. The value is invalid if it is negative.
- `pack` – a `Pack` object representing whether the dinosaur belongs to a pack or not. A dinosaur who is not part of a pack will have this variable set to `null`.

Constructors:
- A constructor that takes in the `name, height, width, weight, speed,` and `pack.`
- A constructor that takes in the `name` and `height` and defaults the `speed` to 30, `pack` to `null`, and all other variables to their default in `Dinosaur`.
- A copy constructor that deep copies all mutable instance variables of the old object to the new object.

Methods:
The following methods should be public.
- `enclosureSize()`
    - Returns a `double` representing the area of the enclosure where the dinosaur is located
    - If the dinosaur is not part of a pack, calculate the enclosure size by multiplying four times the `width` by the `height`.
    - If the dinosaur is part of a `pack`, calculate enclosure size by multiplying the `size` of the `pack` by the `width` by the `height`.
    - This method should override the `enclosureSize()` method from `Dinosaur`.
- `calculateFood()`
    - Returns a `double` representing the amount of food, in pounds, needed to prepare for the dinosaur's meal
    - To calculate, multiply `weight` times the `speed` by the `height`.
- `toString()`
    - Returns the `String` representation of the object.
    - If the dinosaur was not part of a `pack`, return a `String` in the following format: `"<name> requires a <encolosureSize> square foot enclosure and <calculateFood> pounds of food."`
    - If the dinosaur was part of a `pack`, return a `String` in the following format:

```
"<packName> is a family of dinosaurs of size <size>! <name>
requires a <enclosureSize> square foot enclosure and
<calculateFood> pounds of food."
```

- Getters and setters for each of the instance variables.

JurassicPark.java

This Java file is a test driver, meaning it will use all of the above classes! Use this class's main method to test your code.

Methods:

- `main`
  - Create at least 2 `Dinosaur`, 2 `Pterodactyl`, 2 `Velociraptor`, and 1 `Pack`.
  - Use each of the copy constructors at least once. Try modifying the objects to see if you have deep copied properly.
  - Call the relevant `buildEnclosure()` methods on each Dinosaur and its children's objects and print the results.
  - Call `toString()` on all Velociraptors to print the results.
  - These tests and the ones on Gradescope are by no means comprehensive, so be sure to create your own!

Checkstyle

You must run Checkstyle on your submission (to learn more about Checkstyle, check out cs1331-style-guide.pdf under the Checkstyle Resources module on Canvas). **The Checkstyle cap for this assignment is 15 points.** This means there is a maximum point deduction of 15. 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 number of points we would take off (limited by the Checkstyle cap). In future assignments 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:

- `Dinosaur.java`
- `Pterodactyl.java`
- `Pack.java`
- `Velociraptor.java`
- `JurassicPark.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 tests 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 the class forum 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 assignment. 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. **We reserve the right to hide any or all test cases, so you should make sure to test your code thoroughly against the assignment's requirements.**

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.

Burden of Testing

You are responsible for thoroughly testing your submission against the written requirements to ensure you have fulfilled the requirements of this assignment.

Be **very careful** to note the way in which text output is formatted and spelled. Minor discrepancies could result in failed autograder cases.

If you have questions about the requirements given, reach out to a TA or Professor via the class forum for clarification.

Allowed Imports

To prevent trivialization of the assignment, you may not import any classes for this assignment.

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our autograder. 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 assignment at a conceptual high level is allowed. You can discuss course concepts and 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. 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.

The only code you may share are test cases written in a Driver class. If you choose to share your Driver class, they should be posted to the assignment discussion thread on the course discussion forum. We encourage you to write test cases and share them with your classmates, but we will not verify their correctness (i.e., use them at your own risk).

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.