CS 235 - Week 4 Lab Exercise - 2021-09-17

Deadline

Due by the end of lab on 2021-09-17.

How to submit

Submit your `.java` files for this lab on https://canvas.humboldt.edu

Purpose

To practice a bit with redefining an inherited method, writing a class implementing an interface, writing a subclass, and writing some `try-catch` blocks.

Important notes

- IF you are attending the lab via Zoom, you are expected to pair program in a breakout room (possibly trio-program if necessary based on class members' Internet and the number of class members attending via Zoom).
  - In this case, **be sure to TYPE BOTH (all) OF YOUR NAMES** in the beginning comment of EACH of your `.java` files

  But, because of the delta variant surge, if you are attending lab in person in BSS 317, you will each work on a separate computer, although discussion amongst those attending will be encouraged!

Lab Exercise set-up

- FIRST: in the CS50 IDE, in your folder for today's lab exercise, create a local copy of:
  - `GameDie.java` from the Week 1 Lab
  - `Plottable.java`, included along with this lab exercise handout

Problem 1

We have discussed how every Java class that is not explicitly declared as a subclass of another class is a subclass of Java's `Object` class.

And the `Object` class defines a method `toString`, described as follows in the Java API:

```
public String toString()
Returns a string representation of the object.
```

But, the `String` returned by the `toString` method that `GameDie` inherits from the `Object` class is not very descriptive, as we have seen in class!

- So, in your copy of `GameDie` for today's lab, add an `@author` line to your `GameDie` class's opening comment saying that this version is adapted by you/your pair (and give your name/names)
  - And also update the `@version` line in that opening comment, changing the date to today's date.
- Then add a **redefined** version of method `toString` for `GameDie`, to replace its inherited version.
  - Be sure to give it a javadoc-style comment, including a `@return` line.
  - The `String` returned by your `GameDie` class' redefined version of method `toString` should include the number of sides and the current top value of the calling `GameDie` object.

  - It should return a `String` containing all of the calling `GameDie`'s (non-constant) data field values that matches this output format:

    ```
    GameDie[numSides: 6, currTop: 1]
    ```

- Now, to demonstrate this, write a small Java application class `DemoDie.java` whose `main` method contains at least the following (and you may have more than this if you'd like!):
  - create at least one `GameDie` object of your choice.
  - call `System.out.println` with an argument of JUST that `GameDie` object
  - roll that `GameDie` object at least once, printing to the screen the result of that roll
  - again call `System.out.println` with an argument of JUST that `GameDie` object

Problem 2

Fun Facts: the Java `Math` library has `static` methods `sqrt` (to compute a square root) and `pow` (to raise a given number to a given power).

Along with this lab exercise is a Java interface, `Plottable`, in a file `Plottable.java`. (It turns out an interface is pretty straightforward to write!)

Write a public class `Point` that implements this interface `Plottable`, and also meets the following requirements. It should include:

- at least private data fields for a point's x-coordinate, y-coordinate, and name.
- a no-argument constructor that, when called, creates a point with x-coordinate of 0, y coordinate of 0, and name of `""` (the empty string).
- a 3-argument constructor that, when called with an initial x-coordinate, initial y-coordinate, and an initial name, creates a point with those coordinates and that name.
- implementations of the methods required by the `Plottable` interface.
- a mutator `setX` that changes the calling point's x-coordinate
- a mutator `setY` that changes the calling point's y-coordinate
- a mutator `setName` that changes the calling point's name
- a redefined version of the method `toString` inherited from the `Object` class, that returns a `String` containing all of the calling `Point`'s data field values that matches this output format:

  ```
  Point[x: 0.1, y: 13.2, name: finish]
  ```

  - (Note: whatever format the `double` x- and y-coordinates happen to appear by default is fine here!)
- Now, to demonstrate this, write a small Java application class `DemoPoint.java` whose `main` method contains at least the following (and you may have more than this if you'd like!):
  - create at least two `Point` objects of your choice, calling each of its constructors at least once
  - call `System.out.println` at least twice, each time with one of your `Point` objects as its ONLY argument, for each of your `Point` objects
  - call each of `Point`'s mutator methods at least once, actually changing the state of the calling `Point`
  - call method `distFrom` at least once, printing its returned result to the screen in a descriptive message

  - again call `System.out.println`, once for each `Point` object for which you called a mutator method, each time with that `Point` object as its ONLY argument

[Problem 3 and Problem 4 were moved to Homework 3, because only a few reached them.]

- When you are done, or before you leave lab, use Gmail to
  - MAIL a copy of your `.java` files to BOTH/ALL of you, and
  - EACH of you should SUBMIT the required files on Canvas