Homework 4 – Pirate Adventure

Authors: Andrew, Kevin, Lucas, Dhruv, Tejas, Jack

Purpose

Note: This text just contains a fun backstory, not any implementation details. Feel free to skip to the Solution Description and File descriptions.

You are a pirate searching for buried treasure on an island. You and your crew have just reached a ravine, and according to your map, the treasure is right on the other side! However, the bridge to get across the ravine is broken. Fortunately, a wizard who is part of your crew notices a stack of magical wooden planks nearby. They realize that a bridge will be magically formed from these planks if they are placed in order based off color. Since you are well-versed in computer science theory, you realize you can use a sorting algorithm to accomplish this. But after you finish sorting, the bridge still doesn't work! The wizard points out another stack of planks, from which you need a plank of a specific color to complete the bridge. You then realize you can use a searching algorithm to accomplish this! After this, the bridge works, your crew crosses the ravine, and you find the buried treasure!

In this homework you will work on implementing the `Comparable` interface, 2 sorting algorithms and 2 searching algorithms.

Solution Description

For this assignment, you will be modifying a provided `Plank` class. You will also be creating a class called `Pirate`, and a subclass of it called `StrongPirate`. `Pirate` will have 1 sorting and 1 searching method, and `StrongPirate` will override each of these in order to perform a different sorting and searching algorithm.

`Plank.java`

This File represents a `Plank` object. This is a provided file, and we have already given you much of the logic for this file, including constructors, variables, and a complicated `getHue` method. There are a few things you should know about this file before proceeding to `Pirate.java`. You are also responsible for implementing a few things in the file.

Provided: Do not modify any of the following:

- Variables
  - `String woodType`
    - This represents the type of wood of the current Plank (oak, mahogany, etc.)
  - `3 int` variables named `red`, `green`, and `blue`
    - These variables represent the Red, Green, and Blue pixel values for the color of this `Plank` object.
    - These take on values between 0 and 255 (inclusive for both 0 and 255).
  - **Do not modify any of these variables.**
- Constructors
  - The `Plank` class has exactly one 4-argument constructor that initializes its 4 instance variables (see above) accordingly. It takes in the values for woodType, red, green, and blue in that order.
  - **Do not modify this constructor.**
- Methods
  - `getHue()`

- This is a provided method to obtain the hue for the current `Plank` object
- **Do not modify this method.**

You should add the following to Plank.java:

- Interfaces
  - The `Plank` class *must* implement the `Comparable<Plank>` interface.
- Methods
  - `int compareTo(Plank other)`
    - Since the `Plank` class implements the `Comparable<Plank>` interface, it must have a `compareTo` method with this exact signature.
    - This method should return a negative number if the current Plank's hue is less than the other Plank's hue, 0 if their hues are equal, and a positive number if the current Plank 's hue is greater than the other Plank's hue
      - Think about what a simple arithmetic operation between 2 integers may be which has this behavior
    - Note that a Plank object's hue can be obtained using the provided `getHue` method.

Other Notes about Plank.java

Although neither are required, feel free to create an `equals` or `toString` method for the `Plank` class to help you with your testing.

In addition, here is a list of Plank objects that are sorted in hue order to help you test out the functionality of the sorting and searching methods to be discussed in the next 2 files:

```
Plank redPlank = new Plank("Red Wood", 255, 0, 0);
Plank orangePlank = new Plank("Orange Wood", 255, 127, 0);
Plank yellowPlank = new Plank("Yellow Wood", 255, 255, 0);
Plank greenPlank = new Plank("Green Wood", 0, 255, 0);
Plank cyanPlank= new Plank("Cyan Wood", 0, 255, 255);
Plank bluePlank = new Plank("Blue Wood", 0, 0, 255);
Plank purplePlank = new Plank("Purple Wood", 255, 0 255);
```

`Pirate.java`

This file represents a `Pirate` object. This is a file that **you will create**, and you will fill in various methods that implement sorting and searching functionality. `Pirate` should be a class.

- Variables
  - `name` – represents the name of this Pirate, should be of type `String`
- Constructors
  - The `Pirate` class has exactly one 1-argument constructor that takes in a `String`.
- Methods
  - `createBridge(ArrayList<Plank> planks)`
    - Should sort the `planks ArrayList` variable from "least" to "greatest".
    - Because a `Pirate` is not very strong, they can only pull planks from the top of the stack. Therefore, this method should use the **Insertion Sort** algorithm. Specifically, it should be the version of insertion sort where the Planks with the smallest hues are shuffled to the front.

        - You should use `Plank`'s `compareTo` method to determine if one `Plank` object is "less than" another `Plank` object. For example, if we have `Plank` objects `plank1` and `plank2`, and `plank1.compareTo(plank2)` is negative, then `plank1` is considered less than `plank2`.
        - Should not return anything
    - `completeBridge(Plank plankToFind, ArrayList<Plank> planks)`
        - Because a `RegularPirate` is not very strong, they can only find a specific colored plank in the stack one-by-one. Therefore, this method should perform the **Linear Search** algorithm to find a `Plank` object in `planks` that is equal to the `plankToFind` variable.
        - If there is not a `Plank` object in the `ArrayList` equal to `plankToFind`, then return `null`. Otherwise, return the `Plank` object **from the `ArrayList`.**

`StrongPirate.java`

This file represents a stronger `Pirate`. This is a file that you will create, and you will fill in various methods that implement sorting and searching functionality. `StrongPirate` should be a class, and it should **extend the `Pirate` class.**

- Constructors
    - The `StrongPirate` class has exactly one 1-argument constructor that takes in a `String`. It should pass this variable to the `Pirate` superclass's constructor via the `super` keyword.
- Methods
    - `createBridge(ArrayList<Plank> planks)`
        - Should sort the `planks ArrayList` variable from "least" to "greatest".
        - Because a `StrongPirate` is strong, they can pull out Planks from anywhere in the stack. Therefore, this method should use the **Selection Sort** algorithm. Specifically, it should be the version of the selection sort algorithm in which the Plank with the 1st smallest hue is first put at index 0, then the one with the 2nd smallest at index 1, etc.
        - You should use `Plank`'s `compareTo` method to determine if one `Plank` object is "less than" another `Plank` object. For example, if we have `Plank` objects `plank1` and `plank2`, and `plank1.compareTo(plank2)` is negative, then `plank1` is considered less than `plank2`.
        - Should not return anything
    - `completeBridge(Plank plankToFind, ArrayList<Plank> planks)`
        - Because a `StrongPirate` is strong, they will be able to look through the stack efficiently and pull out the correct `Plank` from anywhere in the stack. Therefore, this method should perform the **Binary Search** algorithm to find a `Plank` object in `planks` that is equal to the `plankToFind` variable.
        - Since the Binary Search algorithm only works on sorted lists, you can assume that `planks` is sorted. **Please do not sort `planks` in your completeBridge implementation; you will lose many points if you do this.**
        - If there is not a `Plank` object in the `ArrayList` equal to `plankToFind`, then return `null`. Otherwise, return the `Plank` object **from the `ArrayList`.**

Import Restrictions:

You may only import `java.util.ArrayList`.

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`

Tips & Tricks

As always, feel free to create a driver class to test functionality of the various classes and methods we have asked you to implement. You may also find the provided Game program a fun way to visualize what your code is doing.

Additionally, here are some resources for the various searching and sorting algorithms we have asked you to implement. The solutions here will not work for this assignment as written, but they will give you the general idea of what these algorithms are intended to look like. You may notice some of these solutions use a "recursive" implementation: that is, methods calling themselves. We are not asking you to use recursion in any way, and in fact using recursion would probably overcomplicate the assignment. However, feel free to create local helper methods if you feel you need them.

- https://www.geeksforgeeks.org/binary-search/
- https://www.geeksforgeeks.org/insertion-sort/
- https://www.geeksforgeeks.org/selection-sort/

Collaboration

Collaboration Statement

- To ensure that you acknowledge collaboration and give credit where credit is due, **we require that you place a collaboration statement as a comment at the top of *at least one* java file that you submit**. That collaboration statement should say either:
- *I worked on the homework assignment alone, using only course materials.*
- or
- *In order to help learn course concepts, I worked on the homework with [give the names of the people you worked with], discussed homework topics and issues with [provide names of people], and/or consulted related material that can be found at [cite any other materials not provided as course materials for CS 1331 that assisted your learning].*
- Recall that comments are special lines in Java that begin with `//` or with a JavaDoc comments `/** **/`.
- Failure to include this may result in a point deduction on your homework.

Turn-In Procedure

Submission

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

- `Plank.java`
- `Pirate.java`
- `StrongPirate.java`

Make sure you see the message stating "HW04 submitted successfully". From this point, Gradescope will run a basic autograder on your submission as discussed in the next section.

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 last submission: be sure to submit **every file each time you resubmit**.

Gradescope Autograder

For each submission, you will be able to see the results of a few basic test cases on your code. 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. forgetting checkstyle, 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.

Checkstyle and Javadocs

You must run checkstyle on your submission. The checkstyle cap for this assignment is 30. If you don't have checkstyle yet, download it from Canvas. Place it in the same folder as the files you want checkstyle. 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 points we would take off (limited to the amount we specified above). The Java source files we provide contain no Checkstyle errors. In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

Along with this, you will also 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
```

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 Piazza for all official clarifications