CSE8B - Final Exam (Part 3) - WI22

Due: March 15th, 2022 (Tuesday) at 8am

Hello everyone! Welcome to Part 3 for the Final Exam for CSE 8B WI22.

This document details a take-home exam that you will complete over the next few days. You can NOT communicate with anyone about the content of the assignment until you receive your grade. You can message us privately on Piazza, but the course staff will not give programming advice or answer most questions about the problems. If you have technical trouble creating a screencast (detailed below), then please reach out for assistance as soon as possible.

Do not use any online service other than Piazza to ask questions about the assignment. Do not search for, solicit, or use solutions to the problems that you find elsewhere for the exam. These are all violations of academic integrity that students have committed on exams like this in the past.

You can make use of any course notes, online resources about Java and its libraries, Java tools, and so on to complete the exam, including reusing code from class notes.

Outline of Part 3

Part 3 will be worth 100 points and will be split up into 3 tasks as shown below:

Task 1 - Coding (45 points)
Task 2 - Coding (25 points)
Task 3 - Video (30 points)

Submission Checklist

- `Tradable.java` containing all methods for Task 1
- `Cryptocurrency.java` containing all methods for Task 1
- `Stock.java` containing all methods for Task 1
- `TradingAccount.java` containing all methods for Task 1
- `TravelDistance.java` containing all methods for Task 2
- `explanation_part3.mp4` (or another compatible video extension) that has your recording for Task 3

Tasks

For Part 3, you will have 3 tasks to complete.

If you haven't already, please go through the Autograder guide before you start your exam.

NOTE 1: You do not need to worry about style guidelines for Part 3.

NOTE 2: You should not have to use any extra imports than what is already provided. If you have a problem with the Gradescope Autograder, then you should check your import statements (as well as your own compiler errors).

NOTE 3: PLEASE, *PLEASE* CHECK YOUR PACKAGES before submitting your code! Your IDE may have imported extra packages that are not compatible with the Gradescope Autograder.

NOTE 4: YOU MUST PROVIDE YOUR FACE + PICTURE ID DURING THIS PART OF THE EXAM, OR ELSE YOU WILL NOT GET CREDIT FOR THIS PART OF THE EXAM.

Task 1

For Task 1, you will pretend to be a trader of stocks and cryptocurrencies. You will be implementing two types of `Currency`: `Stock` and `Cryptocurrency` - both of which `extend` the `Tradable` abstract class. The Tradable abstract class is used to determine the maximum possible profit given a currency's price history.

You will be using abstract and concrete classes, interfaces, static and non-static methods, and ArrayLists. Furthermore, you do not need to worry about deep copies.

Please download the starter code from here. Ensure that there are 6 files: `Cryptocurrency.java`, `Currency.java`, `Stock.java`, `Tradable.java`, `TradingAccount.java`, and `TradingAccountTester.java`. **You will be implementing several methods within `Cryptocurrency.java`, `Stock.java`, `Tradable.java`, and `TradingAccount.java`, and you will be testing those methods in `TradingAccountTester.java`.** For Task 1, you need to submit `Cryptocurrency.java`, `Stock.java`, `Tradable.java`, and `TradingAccount.java`.

`TradingAccountTester.java`

Within `TradingAccountTester.java` are a few test cases for the TradingAccount class. **As you are implementing `Cryptocurrency.java`, `Stock.java`, `Tradable.java`,** and

`TradingAccount.java`, you should test thoroughly to make sure that your program is functioning as expected. We will not be collecting `TradingAccountTester.java`.

`Currency.java`

The Currency interface defines the behavior for any currencies. Both the Cryptocurrency class and the Stock class will implement this interface.**This file is already completed for you, so you must NOT make any changes to this file.**

`Tradable.java`

The Tradable class is an ABSTRACT class that defines an object that is tradable. Note that the Tradable class implements the Currency class, but Tradable does not implement the required methods, thus Tradable must be an abstract class.

In specific, a tradable object will have an ArrayList of Integers that represent the object's price history which is denoted as `priceHistory`. **Note that `priceHistory.get(i)` is the price on the i^{th} day**. Keep in mind that `priceHistory` can contain no elements (meaning there is no price history), but if `priceHistory` does contain elements, then it will only contain positive integers.

Within this class, we provide you with a single constructor and a pair of getter and setter methods for `this.priceHistory`.

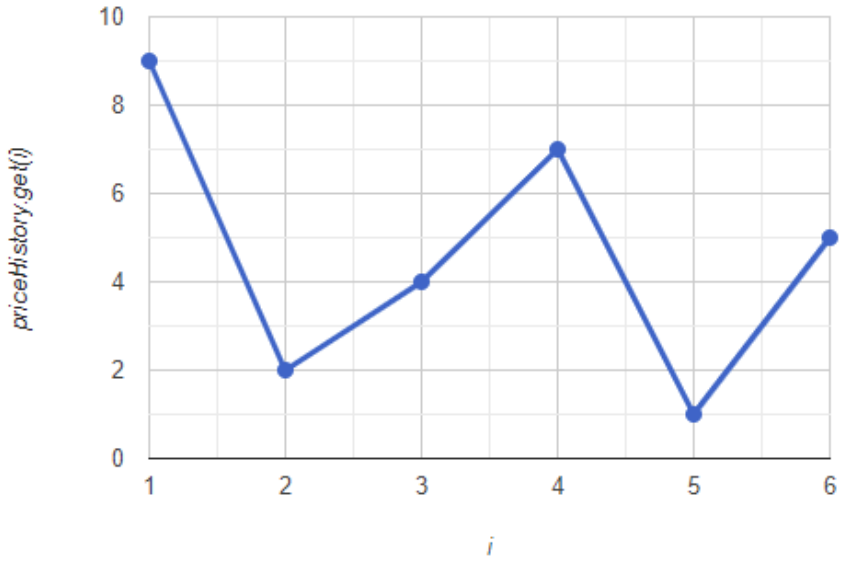
You must implement the following method:

1. `public int getMaxProfit()`

   This method should calculate the maximum amount of profit that can be gained from priceHistory under the assumption that you choose only *a single day* to buy at a given price, then you choose *a different day* in the future to sell at another price.

   For example, consider `this.priceHistory = {9, 2, 4, 7, 1, 5}`. The output of `getMaxProfit()` should be `5` because you would buy on day 2 (where the price is `2`) then sell on day 4 (where the price is `7`). NOTE: You must always buy before you sell. For example, you are unable to buy on day 2 then sell on day 1.

   Below is a visual example:



From the visual example, we can see that if we buy at `i = 2` (where price is `2`), then sell at `i = 4` (where price = `7`), we gain a profit of `5` (`7 - 2 = 5`) which is the maximum profit to be gained.

NOTE: If there is no profit to be *gained*, then this method should simply return `0`. For example, if `this.priceHistory = {8, 4, 3, 2}`, then no matter which day you buy or sell, you cannot yield a positive profit, thus you would return `0`.

HINT: This method can be simply done without anything complex (*i.e.* no need for recursion). At a given index `i`, you should only care about two things: 1. if `priceHistory.get(i)` is the lowest price that you have seen so far (you should declare a variable for this, like `minPrice`), and 2. if `priceHistory.get(i) - minPrice` is the maximum amount of profit that you have seen so far (you should also declare a variable for this, like `maxProfit`).

`Cryptocurrency.java`

The Cryptocurrency class is a CONCRETE class that maintains information about a Cryptocurrency, that `extends` the Tradable class, and that `implements` the Currency interface. **This file contains 1 *private* member variable - `cryptoName`.**

`cryptoName` is simply the name of the cryptocurrency.

You must implement the following 3 methods:

1. `public Cryptocurrency(String cryptoName, ArrayList<Integer> priceHistory)`
   This constructor should set `this.cryptoName` and `this.priceHistory` respectively (**HINT:** use the `super` constructor to set `this.priceHistory`).

2. `public String getName()`
   This method should simply `return this.cryptoName;`

3. `public void setName(String cryptoName)`
   This method should simply set `this.cryptoName` to the parameter `cryptoname`.

`Stock.java`

The Stock class is a CONCRETE class that maintains information about a Stock, that `extends` the Tradable class, and that `implements` the Currency interface. **This file contains 2 *private* member variables - `stockName` and `dateListed`.**

`stockName` is simply the name of the stock, and `dateListed` is the date when the stock was listed publicly where `dateListed` has the format `"YYYY-MM-DD"`. We also provide you with a pair of getter and setter methods for `dateListed`.

You must implement the following 3 methods:

1. `public Stock(String stockName, ArrayList<Integer> priceHistory, String dateListed)`
   This constructor should set `this.stockName`, `this.priceHistory`, and `this.dateListed` respectively (**HINT:** use the `super` constructor to set `this.priceHistory`).

2. `public String getName()`
   This method should simply `return this.stockName;`

3. `public void setName(String stockName)`
   This method should simply set `this.stockName` to the parameter `stockName`.

`TradingAccount.java`

The TradingAccount class represents an account that holds multiple Tradable objects denoted by `tradeHistory`. To be specific, `tradeHistory` (which is `private`) is an ArrayList of either Stocks or Cryptocurrencies.

We provide you with the TradingAccount constructor, so no need to modify the constructor.

You must implement the 2 following methods:

1. `public Tradable tradableMaxProfit()`
   This method should return the Tradable object that yields the maximum profit out of all Tradable objects from `this.tradeHistory`. If there is only one element in

`this.tradeHistory`, then you should return that single Tradable element. Likewise, if there are NO elements in `this.tradeHistory`, then you should simply return null.

For a concrete example, you should refer to the test provided in `TradingAccountTester.java`. In that test, we compare the maximum profit between `appl`, `btc`, and `msft`. Between those three currencies, `btc` would yield the maximum profit, so we return the Tradable object with the name `btc`.

HINT: Think about this method as finding a maximum of some numbers. Use a variable to keep track of the maximum profit that you have seen so far, and use another variable that represents the Tradable object with that maximum profit.

2. `public static Tradable accountsMaxProfit(ArrayList<TradingAccount> accounts)`
   This method should return the Tradable object that yields the maximum profit out of all Tradable objects from out of all the TradingAccount objects from the parameter `accounts`. If there is only one element in `accounts`, then you should return the Tradable object that yields maximum profit out of all the Tradable objects from that one element. Likewise, if there are no elements in `accounts`, then you should simply return null.

   NOTE: There is a possibility that `accounts` could have a TradingAccount that is empty (*i.e.* calling `tradableMaxProfit` on that specific TradingAccount would return `null`). Your program must **NOT** crash in those cases. If all the TradingAccounts are empty, then this program must return null.

   HINT: Once you implement `tradableMaxProfit`, this method should be relatively similar (albeit you will have to handle cases where a TradingAccount's `tradableMaxProfit` could be null).

Task 2

You are given a distance of `n` feet. To travel this distance, you can either take a step (which covers 1 foot) or a leap (which covers 3 feet). Using ***recursion***, you want to calculate the different number of ways you can step or leap to reach a distance of `n` feet.

Please download the starter code from here. Ensure that there are 2 files: `TravelDistance.java` and `TravelDistanceTester.java`. **You will be implementing a single method within `TravelDistance.java` and you will be testing that method in `TravelDistanceTester.java`.** For Task 2, you only need to submit `TravelDistance.java`.

`TravelDistanceTester.java`

Within `TravelDistanceTester.java` are a few test cases for the `TravelDistance` class. Notice how they are surrounded in a `try-catch` block - this implies that the methods must throw some sort of Exception. **As you are implementing `TravelDistance.java`, you should test thoroughly to make sure that your program is functioning as expected.** We will not be collecting `TravelDistanceTester.java`.

`TravelDistance.java`

Within `TravelDistance.java`, you are **required to implement the following method:**

1. `public static int calculateWays(int n) throws Exception`
   This method should ***recursively*** determine the *number of unique ways* you can step (which covers 1 foot) or leap (which covers 3 feet) to reach a distance of parameter `n` feet.

   For example, if `n = 4`, then there are `3` unique ways to reach the distance of `4`:
   1. `step + step + step + step = 1 + 1 + 1 + 1`
   2. `step + leap = 1 + 3`
   3. `leap + step = 3 + 1`

   Hence, if `n = 4`, then you would return 3.

   You should also handle the following edge cases: if `n = 0`, then you should return `1`. if `n < 0`, then you should throw a new Exception with the message `ERR_MSG` (this is a variable given to you).

   HINT: You should first check the edge cases, then call a helper method that only deals with recursion (***you will have to create this helper method from scratch***). Within this helper method, carefully think about your base cases and what needs to be recursively called. For example, if you tried to leap at `n = 2`, then you would go too far (you cannot leap at `n = 2`). That might be a base case to consider, but what should you return in that case? Likewise, how do you recursively call a step and a leap, then combine all results together?

Task 3

For Task 3, you will record a short video of *no more than 4 minutes* that will answer the following two questions. Please read ALL of the instructions and guidelines below before recording your video.

1. In Task 1, consider `this.priceHistory = {7, 2, 5, 1, 6, 9}`. ***Show*** and ***explain*** how your code calculates the maximum profit from that `priceHistory`. In specific, we want to see the following:
    a. Inside `TradingAccountTester.java`, you should create another Stock or Cryptocurrency object with a `name` (and `dateListed`) of your choice that has the `priceHistory` provided above. **We want to see this code written in your tester file.**
    b. *As concisely as you can*, show and explain the logic that happens within `maxProfit` when the method gets called with the provided `priceHistory`. **We want to see not only your `maxProfit` implementation, but we also want to see and hear you actively explain the logic within `maxProfit` given a price history of `{7, 2, 5, 1, 6, 9}`.** This is the most important part!
        i. If your code is not implemented correctly, then you need to still show your incorrect code, but you MUST verbally and concretely explain what the correct logic should be.
    c. Finally, show and describe the maximum profit given that `priceHistory`. **We want to see you run your code, and we see the maximum profit printed to the console.**
        i. If your code does not run correctly, or if the program output is incorrect, then you need to still show the incorrect output, but you also MUST describe what the correct output should be.
    d. We recommend that you spend no more than 1 minute and 30 seconds on the video for Question 1 to save time for Question 2.

2. In Task 2, assume that you did not have to handle any exceptions, and that the recursive implementation is inside `calculateWays` itself (*i.e.* no helper methods). Consider the following single line of code inside the `main` method of `TravelDistanceTester.java`:

```
TravelDistance.calculateWays(3);
```

    For this question, we want you to show and describe a *stack trace* memory model diagram depicting the *recursive* calls to `calculateWays` with an initial input of 3. Because this is a stack frame diagram, we do not need to see any constructor stack frame(s). Furthermore, we expect all respective variables to *start* at the `main` method stack frame (**HINT**: this implies that you should have a stack frame for `main` and a stack frame for `calculateWays(3)`). **It is up to you if you want to have a single stack frame diagram or if you want to have multiple stack frame diagrams, as long as you are able to convey what happens within every stack frame (*i.e.* within every recursive call of `calculateWays`)**.
    a. To be specific, **your stack trace memory model diagram and your explanation should be able to answer the following:**

        i. Within each stack frame (or each call to `calculateWays`), what is the value of parameter `n`?
        ii. At which stack frame(s) do the recursive calls finally stop?
        iii. What values are being returned per stack frame?
        iv. What is the final value returned to the original call inside `main`?
    b. Furthermore, **next to your memory model diagram should be your code for your recursion (most likely, this will be your `calculateWays` helper method).**
        i. Not only is this for our sake, but this code will help you explain your stack trace diagram. **We do NOT know how you implemented this method, so we must see your code next to your diagram.**

An example of a ***stack trace*** memory model diagram can be found inside LecturesWeek9 - 2 - Wednesday.pdf. **You can draw the stack trace memory model diagram in whatever fashion you would like** - you can draw the diagram on a tablet, draw the diagram on paper then take a picture of the drawing, draw the diagram with a (free) drawing program like Google Drawings or GIMP, or you could just use Google Slides to create shapes and texts. **Whatever you do, we just need to see it in the video.**

Here are the required items to include in your video:

1. **For the first few seconds of the video, show ONLY your face and a picture ID** (your UCSD Student ID is highly preferred, but any picture ID with your full name will suffice). Afterwards, you do not have to be on camera, although it is fine to leave it on. This is simply to confirm that it is you recording your video and doing your work. If you do NOT have a webcam, then take a picture of yourself and your picture ID with your phone, and display that picture at the start of your recording.
2. For the rest of the video, answer the 2 questions above *in that exact order*.

Here are some *important* notes in regards to Task 3:

1. **If you do not provide a picture ID, then you will get a 0 on Final Exam - Part 3. YOU MUST PROVIDE YOUR FACE + PICTURE ID, OR ELSE YOU WILL NOT GET CREDIT FOR THIS PRAT OF THE EXAM.**
2. Please be sure to stand as still as possible when showing yourself and your picture ID.
3. Alongside your picture ID, if you do not show your code for Q1 and Q2, then you will also get a 0 on the exam.
4. Video must have sound! Furthermore, while explaining your memory model diagram, make sure to enunciate to the best of your ability. We must hear you clearly explain your diagram! Also, try to NOT use your webcam to show the memory model diagram - that makes your diagram too small to see.

Here is a short tutorial demonstrating how to make a screencast with Zoom: Screencast Tutorial

Submission Instructions

You will be submitting your answers directly to Gradescope under the assignment **"Final Exam - Part 3"**. As a final reminder, **YOU MUST PROVIDE YOUR FACE + PICTURE ID, OR ELSE YOU WILL NOT GET CREDIT FOR THIS PART OF THE EXAM.** The 6 files required for submission are:

- `Tradable.java` containing all methods for Task 1
- `Cryptocurrency.java` containing all methods for Task 1
- `Stock.java` containing all methods for Task 1
- `TradingAccount.java` containing all methods for Task 1
- `TravelDistance.java` containing all methods for Task 2
- `explanation_part3.mp4` (or another compatible video extension) that has your recording for Task 3

If you cannot upload your video to Gradescope:

Gradescope has a max file size of 100MB. Any files larger than that will be rejected. All files must be submitted to Gradescope; we will **not** accept videos (or code) through email or Piazza.

If your video is larger than 100MB, then here are some tips to ensure that your video can be uploaded to Gradescope:

- Use Zoom:
  - A 100MB video is about a 20 minute video on Zoom (1620x1080 video resolution).
  - If you are not using Zoom, then convert your file to an MP4. There are web tools that can do this. MP4 videos have smaller video sizes than other video formats.
- Use a smaller screen size for your computer, or only record a smaller portion of the screen.
- Reduce background clutter (*i.e.* desktop icons). Background clutter reduces compression, making larger file sizes. You can hide the background by maximizing your text editor's window.
- Keep your video short (*i.e.* under 4 minutes):
  - 4 minutes is the max, and it is certainly possible to cover everything in less time.
  - We recommend creating a script of what you are going to say.
  - Pause the video as you switch to the code for the next question (make sure to tell us which question you are showing the code for when you switch).
- Use a 3rd party video editor to reduce the width/height of the video, but make sure your code is not blurry when you play it on Gradescope as we cannot grade blurry videos.