SE 450 - Course Programming Project
DePaul Stock Exchange

Assignment 1

Price Class and Price Factory (100 pts)

Objectives

Create a well-written object-oriented class that represents a Price (i.e., $12.95, $49.99, etc.) and a Price Factory that will be used to create Price objects. You will also create an exception class to handle invalid situations. *ANY exception class you create for our project should extend "Exception" – not "RuntimeException".*

Classes

Price *(put this in a package called "price")*

The Price class will represent a single price value and should contain functionality used with prices (math operations, comparison operations, etc.). Price values should be stored as an *integer* (the number of cents in the price) to avoid decimal precision issues. For example, $12.99 would be 1299 cents, $50.00 would be 5000 cents, etc. *A Price object should be immutable* – once set, the value cannot be changed. Nothing should change the value of an existing Price object. A *new* object should be created to represent a new price value.

The Price class should implement the Comparable<Price> so that it can be value-compared and sorted with other price objects. The interface method *compareTo(Price p)* is described below.

Your Price class should be able to perform the below operations – functionality that we will build upon and make use of later. The required methods, their parameters, return types, and descriptions are as follows:

Note: If null is passed into any of these methods as the Price parameter, throw InvalidPriceException (an exception class that you will create). Any method that overrides a method for the Object class should be annotated using the "@Override" annotation.

- The Price class should have one (and only one) constructor that accepts an int parameter, and that int should be used to set the Price object's *cents* value.
- boolean **isNegative**() ➔ Returns true if the Price value is negative, false if positive or zero.
- Price **add**(Price p) ➔ Returns a new Price object holding the sum of the current price plus the price object passed in. (If the Price parameter passed in is null, you should throw an InvalidPriceException as mentioned earlier).
- Price **subtract**(Price p) ➔ Returns a new Price object holding the difference between the current price minus the price object passed in. (If the Price parameter passed in is null, you should throw an InvalidPriceException as mentioned earlier).
- Price **multiply**(int n) ➔ Returns a new Price object holding the product of the current price and the integer value passed in. Note, as any int value passed in is legitimate, no exception should be thrown by this method.
- boolean **greaterOrEqual**(Price p) ➔ Returns true if the current Price object is greater than or equal to the Price object passed in. (If the Price parameter passed in is null, you should throw an InvalidPriceException as mentioned earlier).
- boolean **lessOrEqual** (Price p) ➔ Returns true if the current Price object is less than or equal to the Price object passed in. (If the Price parameter passed in is null, you should throw an InvalidPriceException as mentioned earlier).

- boolean **greaterThan** (Price p) ➔ Returns true if the current Price object is greater than the Price object passed in. (If the Price parameter passed in is null, you should throw an InvalidPriceException as mentioned earlier).
- boolean **lessThan** (Price p) ➔ Returns true if the current Price object is less than the Price object passed in. (If the Price parameter passed in is null, you should throw an InvalidPriceException as mentioned earlier).
- boolean **equals**(Object o) ➔ Returns true if the current Price object equals the Price object passed in. This overrides "equals" from Object. *(You can use IntelliJ to generate this method for you).* NOTE: This method cannot throw any exceptions.
- int **compareTo**(Price p) ➔ Return the difference in cents between the current price object and the price object passed in. This overrides "compareTo" from Object. *NOTE: This method cannot propagate any exceptions.*
- String **toString**() ➔ Return a string containing the price value formatted as $d*.cc. For example; $50.00, $12.34, $321.10, etc. This overrides "toString" from Object. *NOTE: This method cannot propagate any exceptions.*
- int **hashCode**() ➔ Returns an integer value (unique per price value), generated by a hashing algorithm. This overrides "hashCode" from Object. *(You can use IntelliJ to generate this method for you).* NOTE: This method cannot throw any exceptions.

PriceFactory *(put this in a package called "price")*

Your PriceFactory class should define 2 public static methods called "makePrice" that create and return a new Price objects. One should accept an integer parameter, the other should accept a String parameter. These static functions should be called using the PriceFactory class, not a PriceFactory object:

```
Price p = PriceFactory.makePrice(___);
```

To ensure the PriceFactory methods are used statically and PriceFactory objects are not created, make the PriceFactory class *abstract*.

The 2 *static* "makePrice" functions should operate as follows:

1) Price **makePrice (int value)**: An integer number of cents should be passed in (i.e., 5000 represents $50.00, 12 represents $0.12, 12345 represents $123.45, etc.). Create and return a new price object using the integer cents passed in. Note, as any int value passed in is legitimate, no exception should be thrown by this method.
2) Price **makePrice(String stringValueIn)**: A String price representation should be passed in - this should be converted to an integer number of cents, then use the number of cents to create and return a new Price object using that integer value. The String parameter's price format is:
    - Price strings can come with one optional "$", and optional commas separating thousands-places.
    - A dollar and cent string (separated by a decimal point) [i.e., `1234.56`, `$1234.56`, `$1,234.56`]
    - A dollar string (with or without a decimal point) [i.e., `1234`, `$1234`, `1234.`, `$1,234.`]
    - The minus sign for negatives comes after the "$" (if the "$" is present).

    Other examples of string prices that should be handled by this constructor can be found in the first part of the provided "main" method.

    If a price string is supplied that is not formatted properly, an InvalidPriceException exception should be thrown. Invalid price formats include:

    - Non-2-digit cents values *(Note – a value with no cents value provided like $123 or $123. is valid)*. One-digit cents ("$12.3"), 3 or more-digit cents ("12.345"), etc. are invalid.
    - More than 1 decimal point present (i.e., $12.34.56)

- Non-numeric characters in the string besides one optional "$" and one optional "-". (i.e., "$123.AA", "A123.44", "—1.23", etc.)
- Empty String ("")

Testing

A class called Main that contains a "main" method will be provided that allows you to test your Price and PriceFactory classes. This class should be added to your project in the "src" folder. You will need to add import statements to the Main class to import your Price and PriceFactory classes. This class should compile cleanly and execute without issue once you do that.

The "main" method will indicate the expected value for each test and what your code has generated. The expected output from the "main" will also be provided.

Project Assistance

If you are stuck on some design or code related problem *that you have exhaustively researched and/or debugged yourself*, you can email me a ZIP file of your entire project so that I can examine the problem.

All emailed assistance requests must include a detailed description of the problem, and the details of what steps you have already taken in trying to determine the source of the problem.

Submissions & Grading

When submitting, you should submit a ZIP file of your entire project so that I can compile and execute it on my end. To reduce the size of the zipped file, please remove the "out" folder from your project (if present) before zipping. (This folder is automatically regenerated each time you compile/run so It's safe to delete)

The following are the key points that will be examined in Project when graded:

- Good object-oriented design & implementation
- Proper use of java language conventions
- Proper object-oriented coding practices
- Correct, accurate application execution

Submissions must reflect the concepts and practices we cover in class, and the requirements specified in this document.

Please review the Academic Integrity and Plagiarism section of the syllabus before, during, and after working on this assignment. All work must be your own.

Late submission up to 4 days late will be accepted, though late submissions will incur a 10% penalty (i.e., 10 points on a 100-point assignment). *No late submissions will be accepted for grading after that 4-day period has passed.*

If you do not understand anything in this handout, please ask. Otherwise, the assumption is that you understand the content.