Assignment 1

Object-Oriented Coding

Price Class and 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.

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 total 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. Price objects should be immutable – once set, the value cannot be changed - instead, a new object would be created to represent a new price value.

Note: 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.

Classes

1) Price

Your Price class should be able to perform the below public operations – functionality that we will build upon and make use of later. The required public 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 InvalidPriceOperation (an exception class that you will create). Any method that overrides a method for the Object class should be annotated using the "@Override" annotation.*

- 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.
- Price **subtract**(Price p) ➔ Returns a new Price object holding the difference between the current price minus the price object passed in.
- Price **multiply**(int n) ➔ Returns a new Price object holding the product of the current price and the integer value passed in.
- boolean **greaterOrEqual**(Price p) ➔ Returns true if the current Price object is greater than or equal to the Price object passed in.
- boolean **lessOrEqual** (Price p) ➔ Returns true if the current Price object is less than or equal to the Price object passed in.
- boolean **greaterThan** (Price p) ➔ Returns true if the current Price object is greater than the Price object passed in.
- boolean **lessThan** (Price p) ➔ Returns true if the current Price object is less than the Price object passed in.

- 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).*
- 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).*
- 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.
- 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.

---

2) PriceFactory

Your PriceFactory class should define 2 public static methods called "makePrice" that create and return new Price objects. One method should accept an *integer* parameter, the other accepts 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*)**: This method accepts an integer number of cents (i.e., 5000 represents $50.00, 12 represents $0.12, 12345 represents $123.45, etc.). This method should create and return a new price object using the integer cents passed in.

2) Price **makePrice(*String stringValueIn*)**: This method accepts a String price representation – this should be converted to an integer number of cents. Then use that integer cents value to create and return a new price object. String parameter price format that must be handled: "[$][-]d*[.cc]" *(this is a generic format explained below, not a Java-based format)*

   - *[$] = optional dollar sign, [-] = optional minus sign, d = dollars – 0 or more digits, c = cents – 0 or 2 digits*

   • Optional commas separating thousands-places *are* allowed, i.e., "1,234.56"

   Examples of string prices that must be handled:

"12.85"	"0.75"	"12345.67"	"1,234,567.89"	"12"	".89"
"-12.85"	"-0.75"	"-12345.67"	"-1,234,567.89"	"-12"	"-.89"
"$12.85"	"$0.75"	"$12345.67"	"$1,234,567.89"	"$12"	"$.89"
"$-12.85"	"$-0.75"	"$-12345.67"	"$-1,234,567.89"	"$-12"	"$-.89"

Testing

Create a class called "Main" and add to that class a "main" method [*public static void main(String[] args) { … }*].

In your "main" method, create Price objects using your PriceFactory's two "makePrice" methods with a variety of integer values (for the method that accepts an int parameter) and a variety of Strings (for the method that accepts a String parameter). Verify that the price objects created accurately reflect the expected price values.

Then in the same method, test the various Price methods to insure they generate the correct responses.

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 1 week late will be accepted, though late submissions will incur a 10% penalty (i.e., 10 points). *No late submissions will be accepted for grading after that 1-week time period has passes.*

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