Course Programming Project

DePaul Stock Exchange (DSX)

Part 1

Price & PriceFactory

Part 1 of the Course Programming project involves the creation of the Price & PriceFactory classes.

Price Class

The Price class will be used to represent the various prices used throughout the application (Order prices, Quote prices, Trade prices, Ticker Prices, etc.). Prices are a fundamental component of a trading system so they will be the first thing we will implement.

Price objects, like String objects, should be *immutable* (unchangeable once created). Methods like "add", "subtract", and "multiply" should never modify the state of a Price object, but instead return a *new* Price object whose state reflects the operation performed.

The challenge in creating the Price class is that you are *not* allowed to use a `float` or a `double` to represent the Price object's dollar and cent value. Java `float` and `double` types are not recommended for representing price values, since they always carry small rounding differences. Appendix A of this document ("Precision Issues Representing Currency Using `float` or `double` Types") explains why the use of `float` and `double` types is not recommended.

Additionally, though using java's `BigDecimal` class to represent dollar and cent value is a potential technique, this approach comes with its own set of significant disadvantages. For the details of these significant disadvantages, please be sure to read the short article "BigDecimal and your Money", at: http://lemnik.wordpress.com/2011/03/25/bigdecimal-and-your-money/.

The technique we will use in the Price class for representing price (currency) values is described below:

- Use a java `long` value to represent the value without a decimal place (you are basically storing price values in "cents" only). A price of $10.99 for example, should be represented as 1099. A price of $5,280.00 should be represented as 528000.

  Advantages: Integer (long, short, etc.) values are represented *exactly*, mathematical operations are easy to handle. Disadvantages: Must convert to "decimal" currency representation when printing or otherwise displaying these values (however most any technique would have this same disadvantage).

Types of Prices:

- **"Limit" Prices**
  When an order or quote is entered with a specific price (i.e., an order to buy 100 IBM at $194.95, or an IBM quote of Buy 100@$194.20 and Sell 200@$194.90), the price is considered a "limit" price. The order or quote will only trade at that price (or better) – trading is "limited" to that price (Quotes always have limit prices). Orders can use either limit prices or "market" prices (described below). An order with a limit price that is not near the current market will be "booked" until the stock trades at that price. Use of a limit price on an order is a bit like specifying that you are willing to spend a specific amount (say $4.00) on a gallon of milk, and no more than that. The purchase price might be less than that, but you are limiting the price to a $4.00 maximum.

- **"Market" Prices**
  A "Market" price (used with Orders) is a price that has no actual value. An order with a "Market" price indicates that the order should immediately trade with the opposite side (Buy with Sell or Sell with Buy) at current market prices. For example, if the current market for IBM was Buy 100@$194.20 and Sell 200@$194.90, then a Buy order with a "market" price would immediately trade with the Sell-side at the current market price of $194.90. Likewise, a Sell order with a "market" price would immediately trade with the Buy-side at the current market price of $194.20. Orders with a "market" price are risky because the order submitted does not know the price at which their order might trade. The advantage though is that it will trade immediately. Use of a market price on an order is a bit like specifying that you want to buy a gallon of milk at whatever price your local market is currently charging. The purchase price might low ($3.75) or it might be high ($5.50) – you are more interested in purchasing the milk immediately rather than waiting for a specific price.

Required Price Class Behaviors:

- *package-visible* constructor: Price(long value) - Creates a Price object representing the provided value. Remember, a long value passed in of 1499 represents $14.99, 12850 represents $128.50, etc. Note that since this constructor is "package-visible" only, Price objects cannot be created by classes/objects that are not declared within the same package.

- *package-visible* constructor: Price() - Creates a Price object representing Market price. Note that since this constructor is "package-visible" only, Price objects cannot be created by classes/objects that are not declared within the same package.

- *public* Price add(Price p) throws InvalidPriceOperation - Adds the value of the Price object passed in to the current Price object's value and returns a *new* Price object representing that sum. This does NOT modify either the current Price object or the Price object passed in. Throws InvalidPriceOperation if either Price is a Market Price.

- *public* Price subtract(Price p) throws InvalidPriceOperation - Subtracts the value of the Price object passed in from the current Price object's value and returns a *new* Price object representing that

difference. This does NOT modify either the current Price object or the Price object passed in. Throws InvalidPriceOperation if either Price is a Market Price.

- *public* Price multiply(int p) throws InvalidPriceOperation - Multiplies the value passed in by the current Price object's value and returns a *new* Price object representing that product. This does NOT modify either the current Price object or the Price object passed in. Throws InvalidPriceOperation if either Price is a Market Price.

- *public* int compareTo(Price p) - Standard compareTo logic: Returns 0 if the argument is a Price whose value is *equal* to the current Price object; returns a value less than 0 if the argument is a Price whose value is *greater* than the current Price object; and returns a value greater than 0 if the argument is a Price whose value is *less* than this string.

- *public* boolean greaterOrEqual(Price p) - Return true if the current Price object is greater than or equal to the Price object passed in (you can make use of compareTo). If either Price is a "market" price, return false.

- *public* boolean greaterThan(Price p) - Return true if the current Price object is greater than the Price object passed in (you can make use of compareTo). If either Price is a "market" price, return false.

- *public* boolean lessOrEqual(Price p) - Return true if the current Price object is less than or equal to the Price object passed in (you can make use of compareTo). If either Price is a "market" price, return false.

- *public* boolean lessThan(Price p) - Return true if the current Price object is greater than the Price object passed in (you can make use of compareTo). If either Price is a "market" price, return false.

- *public* boolean equals(Price p) - Return true if the Price object passed in holds the same value as the current Price object (make use of compareTo)

- *public* boolean isMarket() - Return true if the Price is a "market" price, return false if not.

- *public* boolean isNegative()- Return true if the Price is negative, return false if the Price is zero or positive. If the Price is a "market" price, return false.

- *public* String toString() - Return the String format of Price. Format requirements:
  - Positive value examples: $0.49, $0.75, $10.50, $10,000.50, $2,500,000.00
  - Negative value examples: $-0.49, $-0.75, $-10.50, $-10,000.50, $-2,500,000.00
  - Return the String "MKT" for market prices.

Add/Subtract/Multiply Behavior Examples:

Add:	Subtract:	Multiply:
$10.50 add $14.99 = $25.49	$10.50 subtract $14.99 = $-4.49	$10.50 multiply 2 = $21.00
$14.99 add $-51.52 = $-36.53	$14.99 subtract $-51.52 = $66.51	$14.99 multiply 2 = $28.98
$-51.52 add $0.49 = $-51.03	$-51.52 subtract $0.49 = $-52.01	$-51.52 multiply -3 = $154.56
$0.49 add $-0.89 = $-0.40	$0.49 subtract $-0.89 = $1.38	$0.49 multiply 10 = $4.90
$-0.89 add $10.50 = $9.61	$-0.89 subtract $10.50 = $-11.39	$-0.89 multiply 7 = $-6.23

PriceFactory

Price objects should not be created (i.e., "new Price(…)") by classes across the entire application. This would couple other classes to the specifics of Price object creation. Instead, a globally accessible "Factory" called PriceFactory will perform the creation of Price objects i.e., using the Factory design pattern).

When a Price object is needed by some component of the application, all that is needed is a call to the PriceFactory to create a Price object using a provided value:

```
// Create a "limit" price object representing the String "$194.90"
Price myPrice1 = PriceFactory.makeLimitPrice("194.90");

// Create a "limit" price object representing the String held in "someString"
Price myPrice2 = PriceFactory.makeLimitPrice(someString);

// Create a "market" price object - no price value is needed.
Price myPrice3 = PriceFactory.makeMarketPrice();
```

Required PriceFactory Class Behaviors:

- *public static* Price makeLimitPrice(String value) – Creates a (limit) price object representing the value held in the provided String value.
- *public static* Price makeLimitPrice(long value) – Creates a (limit) price object representing the value held in the provided long value.
- public static Price makeMarketPrice() – Creates a (market) price object.

The String formatted values that must be accepted by the "makeLimitPrice(String value)" constructor are listed in detail in Appendix B: "Required String formats for PriceFactory's "String" constructor".

PriceFactory and the "Flyweight" Design Pattern:

Note that in a real-time trading environment, *many* price objects would be created: every order contains a Price object, every quote contains 2 Price objects, other components of this system still to come will also use Price objects.

Real-world financial exchanges regularly receive over a billion quotes a day – which would require 2 billion Price objects to represent. However, there are not actually 2 billion individual price values in use (i.e., every price between $0.01 and $20,000,000.00 would need to be used in trading if the 2 billion prices were all individual unique values). Only a small fraction of those possible prices are really used – the 2 billion price objects would include *many* duplicate values (many users could enter a quote or order that uses the value $10.00 over the course of a trading day).

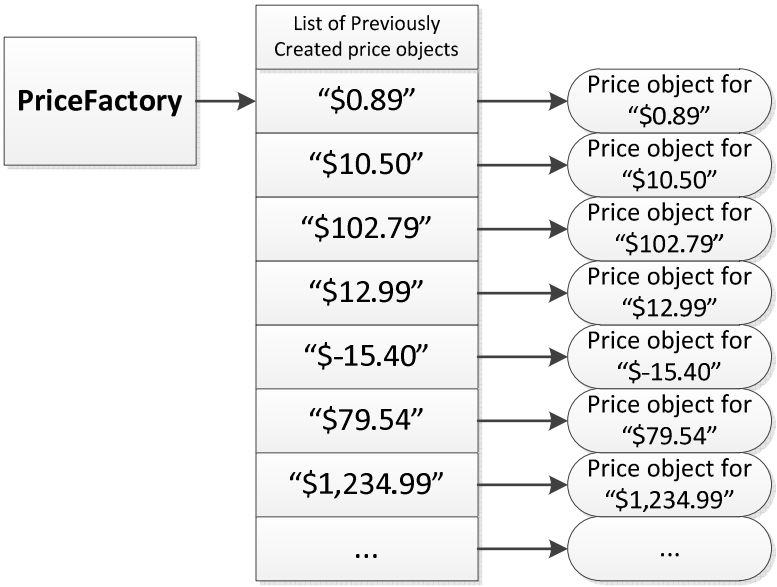
This situation results in the creation of *many* price objects that represent an identical value. Why create 2 billion Price objects when there might only be 2,000 unique price values in use? To remedy this problem, the PriceFactory will implement the "Flyweight" design pattern. The flyweight pattern states that to avoid the creation of a large number of objects containing the identical state (value), create one immutable object with those values and have the components of the system *share* that object. In short – any order or quote that needs to use the value $194.90 will not need to maintain their own personal Price object holding the value $194.90, they can refer to one shared Price object holding the value $194.90.

Flyweight Design Pattern Implementation:

The PriceFactory should support the Flyweight Design Pattern by maintaining a list of all Prices it has previously created. When a request is made to the PriceFactory to create a price object for a certain value, the PriceFactory should *first* check to see if it has previously created a Price object with that value. If it has previously created a Price object with that value, it simply returns a reference to that previously created Price object. If not, it creates a new price object representing the requested value, puts that object in its list of previously created prices, and then returns the desired Price object to the caller.

Example:

If the PriceFactory (shown below) was asked to create a Price object for the value "$12.99", it would *first* check to see if it has previously created a Price object with that value. Since it has ("$12.99" *is* in the list of previously created Price objects), it would simply return a reference to that previously created Price object. If the PriceFactory was asked to create a Price object for the value "$13.75", it would *first* check to see if it has previously created a Price object with that value. Since it has not ("$13.75" *is not* in the list of previously created Price objects), it would create a new price object representing "$13.75", put that Price object in the list of previously created prices, and then return the Price object to the caller.



Testing Phase 1

A test "driver" class with a "main" method will be provided that will exercise the functionality of your Price & PriceFactory classes. This will not exhaustively test your classes but successful execution is a good indicator that your classes are performing as expected. The output of the test driver is shown below:

```
Creating some Test Price Objects.
Verifying the Values in Your Test Price Objects:
$10.50    -->    $10.50 : CORRECT
$1,400.99 --> $1,400.99 : CORRECT
$-51.52   -->   $-51.52 : CORRECT
$0.49     -->     $0.49 : CORRECT
$-0.89    -->    $-0.89 : CORRECT
$12.00    -->    $12.00 : CORRECT
$90.00    -->    $90.00 : CORRECT
$14.50    -->    $14.50 : CORRECT
MKT       -->       MKT : CORRECT

Verifying the Functionality of your Mathematical Operations:
$10.50    + $1,400.99 = $1,411.49 : CORRECT
$1,400.99 - $1,400.99 =     $0.00 : CORRECT
$-51.52   +     $0.49 =   $-51.03 : CORRECT
$0.49     *         4 =     $1.96 : CORRECT
$-0.89    -    $12.00 =   $-12.89 : CORRECT
$12.00    +    $90.00 =   $102.00 : CORRECT
CORRECT: Cannot add a LIMIT price to a MARKET Price: MKT + $10.50
CORRECT: Cannot subtract a LIMIT price from a MARKET Price: MKT - $10.50
CORRECT: Cannot multiply a MARKET price: MKT * 10

Verifying the Functionality of your Boolean Checks:
Value      | Is Negative | Is Market
-----------------------------------
$10.50     | CORRECT     | CORRECT
$1,400.99  | CORRECT     | CORRECT
$-51.52    | CORRECT     | CORRECT
$0.49      | CORRECT     | CORRECT
$-0.89     | CORRECT     | CORRECT
$12.00     | CORRECT     | CORRECT
$90.00     | CORRECT     | CORRECT
$14.50     | CORRECT     | CORRECT
MKT        | CORRECT     | CORRECT

Verifying the Functionality of your Boolean Comparisons:
Comparison
to $14.50  | greaterOrEqual  | greaterThan  | lessOrEqual  | lessThan
-------------------------------------------------------------------------
$10.50     | CORRECT         | CORRECT      | CORRECT      | CORRECT
$1,400.99  | CORRECT         | CORRECT      | CORRECT      | CORRECT
$-51.52    | CORRECT         | CORRECT      | CORRECT      | CORRECT
$0.49      | CORRECT         | CORRECT      | CORRECT      | CORRECT
$-0.89     | CORRECT         | CORRECT      | CORRECT      | CORRECT
$12.00     | CORRECT         | CORRECT      | CORRECT      | CORRECT
$90.00     | CORRECT         | CORRECT      | CORRECT      | CORRECT
$14.50     | CORRECT         | CORRECT      | CORRECT      | CORRECT
MKT        | CORRECT         | CORRECT      | CORRECT      | CORRECT
```

```
Verifying your Flyweight Implementation:
Price $10.50    is same object as new    $10.50: CORRECT
Price $1,400.99 is same object as new    $10.50: CORRECT
Price MKT       is same object as new       MKT: CORRECT
Price $1,400.99 is same object as new       MKT: CORRECT
```

Phases & Schedule

The Course Programming Project will be implemented in phases, each with a specific duration and due date as is listed below. Detailed documents on each phase will be provided at the beginning of the phase.

Phase 1 (1 Week) 9/17 – 9/23:

- **Price & Price Factory**

Phase 2 (1 Week) 9/24 – 9/30:

- Tradable & Tradable DTO
- Order
- Quote & Quote Side

Phase 3 (2 Weeks) 10/1 – 10/14: [Midterm 10/8]

- Current Market Publisher
- Last Sale Publisher
- Ticker Publisher
- Message Publisher
- Fill Message
- Cancel Message
- Market Message
- User (interface)

Phase 4 (2 Weeks) 10/15 – 10/28:

- Product Service
- Book & Book Side
- Trade Processor

Phase 5 (1 Week) 10/29 – 11/4:

- User Implementation
- User Command Service

Phase 6 (1 Week) 11/5 – 11/12:

- User Interface GUI
- Simulated Traders

[Final Exam 11/19]

Appendix A: Precision Issues Representing Currency Using `float` or `double` Types

While floating-point data types are capable of representing extremely large positive and negative numbers and offer the equivalent of many decimal digits of precision, they are nonetheless inexact when it comes to representing decimal numbers. The reason for this stems from the fact that computers are binary, not decimal, machines. Internally, the hardware must represent a decimal number using a binary format, and most real decimal numbers cannot be exactly represented in a fixed-length binary format.

For example, the decimal value 9.48 cannot be represented exactly as a binary floating-point value; it must be represented as a close approximation – in this case, 9.479999542236328125. Fortunately, Java's built-in float-to-String conversion methods can identify such an approximation, and display the value as 9.48, instead of 9.479999542236328125, as can be seen in the below example:

```
float unitCost = 9.48f;
System.out.println("Value: " + unitCost);
```

Generates `Value: 9.48`

Now, suppose you wish to compute the total cost of 100 items having a unit cost of $9.48:

```
float unitCost = 9.48f;
float totalCost = unitCost * 100.0f;
System.out.println("Total Cost: $" + totalCost);
```

Generates `Total Cost:  $947.99994`

In this case, the effects of inexactness when using a floating-point data type to represent a monetary value are obvious. This example indicates that 9.48 * 100.0 is not 948.0 as you would expect, but rather 947.99994. If that computed value is then truncated to two decimal places, the result is $947.99, not $948.00. It would cost you a penny if you were preparing a customer invoice – every time this happens. The problem only gets worse as the computations become more complex.

Appendix B: Required String formats for PriceFactory's "String" constructor

Valid formats for the String passed to the "makeLimitPrice(String value)" constructor are shown below:

(*"d" = dollar values, "c" = cent values)*

- *,d.c*, **$*,**d.c*
- **-*,**d.c*, **$-*,**d.c*, **-$*,**d.c*
- .c*, $.c*, -.c*, $-.c*, -$.c*
- *,d[.], $*,d[.], -*,d[.], $-*,d[.], -$*,d[.]

✓ *"**,**d" indicates any number of dollar-value digits, that <u>may or may not</u> have commas every 3 digits (i.e., 12.34 or 12345.67 or 12,345.67)*

✓ *".c*" indicates one or more digit cents amount. A single-digit cents about should be assumed to have a trailing zero (i.e., ".6" should be equivalent to ".60", ".1" should be equivalent to ".10", etc.). NOTE: if more than 2 cents-digits are provided, any digits past the 2nd cents digit will be truncated and not considered in the value. (i.e., $12.165 should be equivalent to $12.16, $50.123456 should be equivalent to $50.12).*

Example values for Required Formats:

- *,d.c* "0.25", "1.49", "2234.56", "2,432.67", "14.5"
- $*,d.c* "$0.25", "$1.49", "$2234.56", "$2,432.67", "$14.5"
- -*,d.c* "-0.25", "-1.49", "-2234.56", "-2,432.67", "-14.5"
- $-*,d.c* "$-0.25", "$-1.49", "$-2234.56", "$-2,432.67", "$-14.5",
- -$*,d.c* "-$0.25", "-$1.49", "-$2234.56", "-$2,432.67", "-$14.5"
- .c* ".49", ".70", ".6", ".255" (Note, any digits past the 2nd cents digit will be truncated)
- $.c* "$.49", "$.70", "$.6", "$.255" (Note, any digits past the 2nd cents digit will be truncated)
- -.c* "-.49", "-.70", "-.6", "-.255" (Note, any digits past the 2nd cents digit will be truncated)
- $-.c* "$-.49", "$-.70", "$-.6", "$-.255" (Note, any digits past the 2nd cents digit will be truncated)
- -$.c* "-$.49", "-$.70", "-$.6" , "-$.255" (Note, any digits past the 2nd cents digit will be truncated)
- *,d[.] "50", "200", "15700", "5,280", "50."
- $*,d[.] "$50", "$200", "$15700", "$5,280", "$50."
- -*,d[.] "-50", "-200", "-15700", "-5,280", "-50."
- $-*,d[.] "$-50", "$-200", "$-15700", "$-5,280", "$-50."
- -$*,d[.] "-$50", "-$200", "-$15700", "-$5,280", "-$50."

Sample Algorithm for Converting String-Formatted Price Values to Long Values: