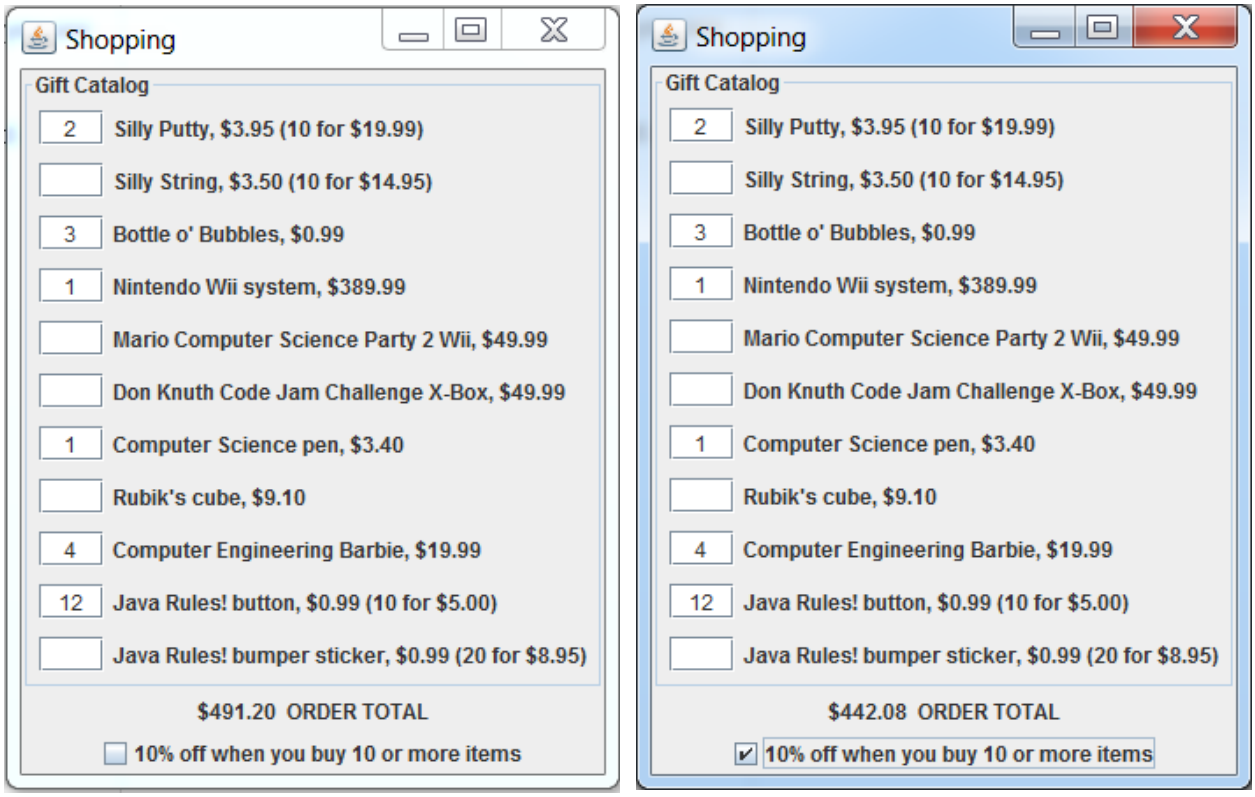
EGR326 Software Design and Architecture
Homework #1 Shopping Cart
Due: 1/13/2017 5 PM

This program focuses on review of Java, writing a multi-class object-oriented program, and collections You will need to download the support files HERE. Your project will run the GUI.
Run ShoppingMain to launch the program. Note that input files like catalog.txt should be placed in your IntelliJ project's root folder, the parent of its src/ and the two Java files should be placed under src/.

We will revisit this program in future assignments to modify and improve the code and spec for the classes written here.

Program Description:

In this program you will write a set of supporting classes for a basic shopping program. You are given the Graphical User Interface that will provide the "front end" to your program. You are to write the back end (what is sometimes referred to as the "domain specific code"). The program displays a catalog of items that can be purchased. You can purchase more than one of a given item. Prices are expressed using real numbers (doubles). Quantities to purchase are expressed as integers (ints). For example, you can't buy 2.5 of something.
Some items have a discount when you buy enough of them. For example, Silly Putty normally costs $3.95 each, but you can buy 10 for $19.99. These items have, in effect, two prices: a single item price and a bulk item price for a bulk quantity.



When computing the price for a discounted bulk item, apply as many of the bulk quantity as you can and then use the single item price for any leftovers. For example, the user is ordering 12 buttons that cost $0.99 each but can be bought in bulk 10 for $5.00. The first 10 are sold at that bulk price ($5.00) and

the two extras are charged at the single item price ($0.99 each) for a total of $6.98. If the user had bought 32 buttons, the first 30 would cost $15.00 (3 * $5.00) and the two extras would be $0.99 each for a total of $16.98.

At the bottom of the window is a checkbox for an overall discount. If this box is checked, the user is given a 10% discount off the total price only if the cart contains at least 20 total items. If the cart contains less than 20, even if this box is checked, the order total should not apply the discount. The discount is computed using simple double arithmetic, computing a price that is 90% of what it would be otherwise if the quantity is large enough. If the box is not checked then the 10% discount should not be applied regardless of the total number of items.

Classes to Implement:

Your task is to implement the following classes that are used to make this code work:

- `Item`, a single item that can be purchased
- `DiscountedItem`, a single item with a bulk discount for high-quantity purchases
- `Catalog`, a set of all items that are available in the store
- `Purchase`, a single item to be purchased in a given quantity
- `ShoppingCart`, the list of all purchases that the user currently wants to make
- `ShopTest`, a JUnit test class to verify the functionality of the other classes

For full credit, all methods of all classes should run in a constant amount of time (O(1)) regardless of any parameter value(s) passed, unless otherwise specified. This constraint may affect your choice of implementation and data structures.

Item class:

An `Item` object stores information about an individual item. It should have the following public behavior:

Method	Description
`Item(`*name, price*`)`	Constructor that takes the item's name (a string) and its price (a real number) as arguments.
`getName()`	Returns the name of the item as passed to the constructor.
`priceFor(`*quantity*`)`	Returns a real number representing the price for a given quantity of the item (an integer).
`toString()`	Returns a text representation of this item: its name followed by a comma and space followed by its price. The price should be formatted properly as dollars and cents. For example, your method might return `"Rubik's cube, $9.10"`. This is non-trivial because prices that have a different number of digits after the decimal point. The method `String.format` can help you to format a real number with exactly 2 digits after the decimal point, or you may use the `NumberFormat.getCurrencyInstance` method to achieve the same effect. See the Java API documentation.

There should be only two fields, in particular, `name` and `price` for this class. `price` and `name` should be updated only in the constructor. `price` field should store the price of a single item. `priceFor` method should not change `price` field but simply calculate the price for the quantity given as the parameter.

DiscountedItem class:

The `DiscountedItem` class is a subcategory of `Item`. A `DiscountedItem` object stores information about an individual item that has a bulk discount when purchased in sufficient quantity. It should have the following public behavior in addition to the behavior of `Item`.

Method	Description
`DiscountedItem(`*name*`,` *price*`,` *bulk quantity*`,` *bulk price*`)`	Constructor that takes a name (a string), a single-item price (a real number), a bulk quantity (an integer), and a bulk price (a real number) as arguments.
`toString()`	Returns a text representation of this discounted item, consisting of its name, followed by a comma and space, followed by its price, followed by an extra space and a parenthesized description that has the bulk quantity, the word "for" and the bulk price. For example, your method might return `"Silly Putty, $3.95 (10 for $19.99)"`.

In addition, you need to override `priceFor` method such that it returns the discounted price based on the quantity. Say we have a *`button`* object which costs cost $0.99 each but can be bought in bulk 10 for $5.00. Thus, this button object is of `DiscountedItem` type. `button.priceFor`(9) should return 9 * 0.99 (the same as the parent). `button.priceFor`(23) should return 2 * 5 + 3 * 0.99. (20 out of 3 were bulk price and the remaining 3 had the original price).

Catalog class:

A Catalog object stores information about all items available for purchase. It has the following public behavior:

Method	Description
`Catalog(`*storeName*`)`	Constructor that takes a store name (a string) as its argument. The catalog is initially empty.
`add(`*item*`)`	Adds the given item to the end of this catalog's collection of items.
`getItem(`*name*`)`	Returns the item, if any, whose name exactly matches the given name. You may assume that an item has been previously been added to the catalog that has the given name. The behavior of this method is unspecified if no such item exists in the catalog.
`getStoreName()`	Returns the catalog's store name as passed to the constructor.
`iterator()`	Returns an iterator over all items in the catalog. The iterator should return the items in the order that they were originally added to the catalog. (Don't implement your own iterator class from scratch; ask your catalog's internal data structure for its iterator and return that.)

The `Catalog` class should also implement the **`Iterable`**`<Item>` interface from `java.lang` so that a `Catalog` object can be used as the target of a "for-each" loop over its items. (The for-each loop internally calls the `iterator` method on the catalog and asks the iterator for each item.) See the Java API documentation if you are unfamiliar with `Iterable`. Also, read Headfirst Java's Iterator Pattern.

Purchase class:

A `Purchase` object stores information about a purchase order of a particular item: namely, the item itself, and the quantity desired to be purchased. It should have the following public behavior:

Method	Description
`Purchase(`*item*`,` *quantity*`)`	Constructor that creates a purchase for the given item and given quantity (an integer).
`getPrice()`	Returns the cost to purchase the item at the given quantity.
`getQuantity()`	Returns the quantity for this purchase as passed to the constructor.
`isEmpty()`	Returns whether this purchase has a quantity of 0 (`true` if so).
`matches(`*purchase*`)`	Returns whether this purchase is for the same item as the given other purchase (`true` if so, `false` if not).

In addition to above, you may add another public method `updateQuantity(newQuantity)` method which updates the quantity to the new quantity (an integer).

ShoppingCart class:

A `ShoppingCart` object stores information about the customer's overall order, implemented as a collection of `Purchases`. The internal order of the purchases in the cart is unspecified. It should have the following public behavior:

Method	Description
`getDiscountPercentage()`	This *static* method returns the percentage to discount carts that contain enough items (10).
`getDiscountQuantity()`	This *static* method returns the minimum quantity where a discount will apply to carts (20).
`ShoppingCart()`	Constructor that creates an empty shopping cart of purchases.
`add(`*purchase*`)`	Adds a purchase to the shopping cart, replacing any previous purchase for this item with the new purchase. For example, a user at one time might request to purchase 3 of some item and later change the request to purchase 5 of that item. The purchase for 5 replaces the purchase for 3. The user is not requesting 8 of the item in making such a change. The `add` method might be passed a purchase with a quantity of 0. This indicates that the client doesn't want any purchase to be added; instead, the client wants any existing purchase for that item to be removed. This method should run in no worse than O(*n*) time.
`clearAll()`	Removes all purchases from the cart.
`getTotal()`	Returns the total cost of all the purchases in the shopping cart. This method should run in no worse than O(*n*) time.
`hasDiscount()`	Returns whether or not this cart should get a 10% discount when it contains 20 total items or more. Initially `false`, but can be changed to `true` by a call to `setDiscount`.
`setDiscount(`*value*`)`	Sets whether or not this cart should get a 10% discount when it contains 20 total items or more (a parameter value of `true` means there is a discount, `false` means no discount).
`totalQuantity()`	Returns the total quantity from all combined purchases in this cart. This method should run in no worse than O(*n*) time.

You should have a private boolean field indicating whether the discount should be applied and have `hasDiscount` simply return this field. This boolean field should be set to false in the constructor and `setDiscount` method will be the one who changes this field based on the criteria.

ShopTest class:

It's important to test your code incrementally as you are writing it. Therefore you will turn in a ShopTest class that contains a small amount of code to test your other classes. The class may contain any contents you like, so long as it constructs objects of at least 3 of your other classes and calls some of their methods. The GUI does not call every method of every class nor call them with every possible parameter value, so we encourage you to try testing various behavior. Your ShopTest class should be a JUnit test case file. For this HW, you will be also graded on the quality and the coverage of your tests.

Implementation Details:

Do not add any other public methods to these classes, although you can add your own private methods.
You are allowed to define a toString method in any of these classes (you might find that helpful in testing and debugging your code).

Use your **testing code** to develop the classes in stages rather than all at once. When your classes are working, only then combine them with the provided classes to make sure that they work properly.
Assume valid parameters. You may assume that all parameter values passed to all methods and constructors are valid: that prices are always greater than 0, quantities are non-negative, and all objects are non-null.

Your classes are to exactly reproduce the format and overall prices shown in the two example screenshot. You will have to run the GUI and enter the individual quantities from the two screenshots to verify that your classes are working correctly.

Creative Aspect: mycatalog.txt

Along with your program, turn in a file mycatalog.txt that represents another catalog of items to be purchased. The file's format should match the provided catalog.txt, containing one item per line, with name / price separated by commas.

```
Bottle o' Bubbles,0.99
Nintendo Wii system,389.99
```

Discounted items contain the bulk quantity and bulk price on the same line with additional commas. For example:

```
Silly Putty,3.95,10,19.99
Silly String,3.50,10,14.95
```

For full credit, your file should contain at least 8 unique items and at least one of each kind of item (discounted and not).

Style and Design Guidelines:

Some of your classes will use collections internally to store data. Part of your grade will be based on whether you choose appropriate collections to match the expectations outlined in this spec, such as performance and data ordering.

Redundancy is a major grading focus of every assignment for this course. Some methods are similar in behavior or based off of each other's behavior. You must avoid repeated logic as much as possible. Your class may have other (helper) methods besides those specified, but any other methods you add should be private.

You should use **constants** where appropriate to avoid "magic numbers" (fixed values used in your code). Policy data such as specific values used to decide ranges or particular costs are especially good candidates to be made into constants.

Follow **good general style** such as: making fields private, avoiding unnecessary fields (don't declare variables as fields that could be declared locally); initializing fields in constructors, rather than as they are declared; declaring collections using interface types (e.g. `List` rather than `ArrayList`); appropriately using control structures like loops and if/else; properly using indentation, good variable names and types; and not having any lines of code wider than 100 characters.

Comment all of your files descriptively in your own words at the top of each class, each method/constructor, and on complex sections of your code. Comments at the top of a class should identify yourself, the assignment / course / section, and should describe the overall purpose of the class. Method header comments should at a minimum explain the method's behavior, parameters, and return values as appropriate.

For reference, my solution contains roughly 110 "substantive lines" (which excludes things like blank lines, comments, and } brace lines) according to the class Indenter Tool, though this number is just provided as a sanity check; you do not need to match it or be close to it to get full credit.

Grading:

Part of your program's score will come from its "external correctness", or whether your code's behavior matches exactly what is expected. Programs that do not compile receive zero external correctness points.

The rest of your program's score will come from its "internal correctness." Internal correctness measures whether your source code follows the stylistic guidelines indicated in this document.

How to Submit:

Create a **private** repository in GitHub named **EGR326-HW1** (Or **CSC526-HW1** for grad students). Share the repository with your instructor (ID:mikiehan). Push all files in your IntelliJ project to GitHub. After you push, create a screenshot showing the root directory of your GitHub repository. Upload a Word doc containing this screenshot to BB → HWs → HW1.