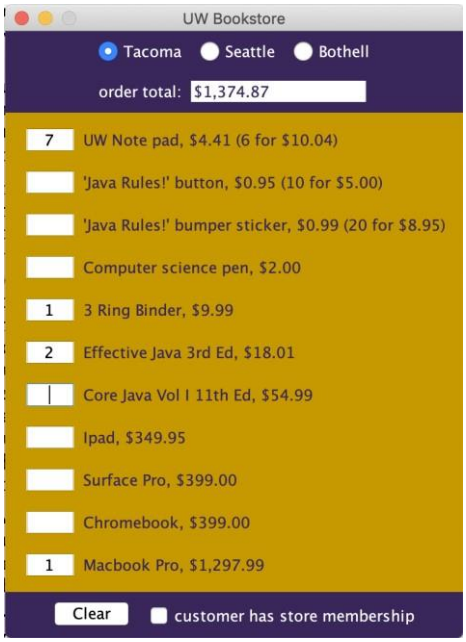
School of Engineering and Technology, University of Washington Tacoma
TCSS 305 Programming Practicum
Assignment 3 – Bookstore
50 points

Part I (half of the assignment grade)

Program Description:

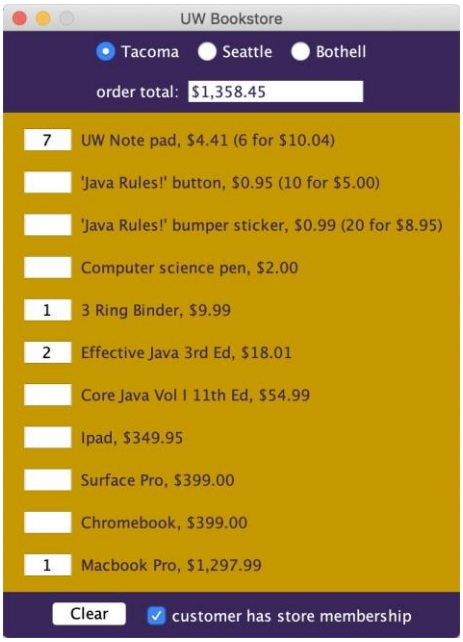
This part of the assignment will give you practice writing Java classes to a provided API and using Collections. You are to write a set of supporting classes for a simple shopping cart. The instructor has provided the Graphical User Interface (GUI) that will provide the "front end" or "view" to your program. You are to write the back end (what is often referred to as the "domain specific code" or the “model”).

Prices are expressed as real numbers and quantities are expressed as integers (*i.e.*, you can’t buy 2.345 units of something). Notice that some of the items have a discount when you buy more. For example, Silly Putty normally costs $4.41 per unit, but you can buy 6 for $10.04. These items have, in effect, two prices: a single item price and a bulk item price for a bulk quantity. The bulk quantity discount is only used if the membership checkbox at the bottom of the GUI is selected. When computing the total with the bulk item discount, 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 $0.95 each but can be bought in bulk at 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 ($0.95 each) for a total of $6.90. If the user were ordering 22 buttons, the total would be $11.90 (two bulk quantities plus two extras).



Here is a screen shot of what the program could look like when the user has selected various items to order (be platform dependent)note that the exact appearance may

At the bottom of the GUI there is a checkbox for a discount for customers who have a store membership. If this box is checked, the bulk pricing discount is applied to any items with bulk pricing as described above.



Note the selected checkbox

Implementation Guidelines:

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

- `Item`
- `ItemOrder`
- `Cart`

An `Item` object stores information about an individual item. It must have the following public methods. Note that the names (such as *price* and *bulk quantity*) given here are not meant to be the actual parameter names (Checkstyle would complain about them); they're meant to be descriptive.

Method	Description
`Item(`name, price`)`	Constructor that takes a name and a price as arguments. The name is a `String` and the price is a `BigDecimal`.
`Item(`name, price, bulk quantity, bulk price`)`	Constructor that takes a name, a single-item price, a bulk quantity, and a bulk price as arguments. The name is a `String`, the quantity is an `int` and the prices are `BigDecimal` s.
`getPrice()`	Returns the single item price for this Item. (BigDecimal return type)
`getBulkQuantity()`	Returns the bulk quantity for this Item. (int return type)
`getBulkPrice()`	Returns the bulk price for this Item. (BigDecimal return type)
`isBulk()`	Returns True if the Item has bulk pricing; false otherwise
`toString()`	Returns a `String` representation of this Item: name, followed by a comma and a space, followed by price. If the item has a bulk price, you should append an extra space and a parenthesized description of the bulk pricing that has the bulk quantity, the word "for" and the bulk price. See the examples below.
`equals(`*object*`)`	Returns `true` if the specified object is equivalent to this Item, and `false` otherwise. Two items are equivalent if they have *exactly* equivalent names, prices, bulk quantities and bulk prices. This method must properly override `java.lang.Object.equals()`.

`hashCode()`	Returns an integer hash code for this item. This method must override `java.lang.Object.hashCode()` and be consistent with `equals()`.

The `String` representation of an Item must *exactly* match that shown in the screenshots. For example, an Item named "X" with a per-item price of $19.99 and no bulk quantity would have the `String` representation "`X, $19.99`" (without the quotes); an item named "X" with a per-item price of $19.99, a bulk quantity of 5, and a bulk price of $89.99 would have the `String` representation "`X, $19.99 (5 for $89.99)`" (without the quotes). The format of these String representations will be tested.

An `ItemOrder` object stores information about a purchase order for an item: namely, a reference to the item itself and the quantity desired. It must have the following public methods.

Method	Description
`ItemOrder(`*item*`,` *quantity*`)`	Constructor that creates an item order for the given quantity of the given `Item`. The quantity is an `int`.
`getItem()`	Returns a reference to the Item in this ItemOrder.
getQuantity()	Returns the quantity for this ItemOrder.
`toString()`	Returns a String representation of this ItemOrder: You may use any format that seems reasonable to you for this String.

A `Cart` object stores information about the customer's overall purchase. One field must be a collection of some type to hold information about all the Items the customer has ordered. (Use a generic collection from the Java Collections Framework. One possible choice, though perhaps not the easiest to work with, is List<ItemOrder>). `Cart` class must have the following public methods.

Method	Description
`Cart()`	Constructor that creates an empty shopping cart.
`add(`*order*`)`	Adds an order to the shopping cart, replacing any previous order for an *equivalent* item with the new order. The parameter is an `ItemOrder`. (`equals()` would return true if used to compare *equivalent* items) The return type of this method is `void`
`setMembership(`*value*`)`	Sets whether or not the customer for this shopping cart has a store membership (the parameter is a `boolean`; `true` means the customer has a membership, `false` means the customer doesn't). The return is `void`.
`calculateTotal()`	Returns the total cost of this shopping cart as a `BigDecimal`. This returned BigDecimal should have scale of 2 and use the ROUND_HALF_EVEN rounding rule.
`clear()`	Removes all orders from the cart. The return is `void`.
`getCartSize()`	Returns the number of ItemOrders currently in the cart.

`toString()`	Returns a String representation of this `Cart`: You may use any format that seems reasonable to you for this String.

You *must not* change any method signatures or return types defined in this assignment. (You may change parameter *names* but not parameter data types.) You *must not* introduce any other non-private methods to these classes, although you may add your own *private* helper methods. You must override `toString` in these classes (you may find this helpful for testing and debugging). You are also allowed to override other (non-`final`) methods declared in `java.lang.Object`, such as `equals` and `hashCode`. If you do, however, your definitions must be reasonable and consistent with each other; if you implement inconsistent `equals` and `hashCode` methods, you will certainly lose points. You *must* override `equals` and `hashCode` for the `Item` class, as described above (this will be tested when I grade your assignment even if you don't use those methods in your own code).

For this assignment, code defensively:

Setters and constructors MUST test for invalid values.

Throw `IllegalArgumentException` (explicitly) for any of the following conditions: prices passed to your classes are < 0, quantities passed to your classes are < 0, Strings passed to your classes are empty.

Throw `NullPointerException` (explicitly) for any of the following conditions: `String`s passed to your classes are `null`, `BigDecimal` objects passed to your classes are `null`, the `Item` passed to the `ItemOrder` constructor is `null`, the `ItemOrder` passed to the `add` method in `Cart` is `null`.

(Of course the behavior for overridden methods from class Object, such as the `.equals()` method, are defined in class Object.)

Hints and Suggestions:

As you develop your classes do not code any dependencies based on the current list of items. That is, your classes should still work correctly if the list of items is replaced with an entirely different list of items.

Most of these methods are simple to write but notice that when you add an `ItemOrder` to a `Cart` you must deal with replacing any old order for an equivalent item. A user might initially request 3 of some item and later change the request to 5 of that item. The order for 5 replaces the order for 3. The user is not requesting 8 of the item when they make such a change. The add method might be passed an item order with a quantity of 0. This should behave just like the others, replacing any current order for the item.

One technique (which is not the best one but will certainly work) is to use an `ArrayList` to implement your `Cart`. If you do, the methods you are most likely to be interested in are the following:

Method	Description
`ArrayList<T>()`	Constructor that creates an empty `ArrayList` to hold elements of type `T`.
`add(`*value*`)`	Adds the given value to end of the `ArrayList`. The parameter type is `T`, so only the type you specified at construction can be added to the `ArrayList`.

`get(`*index*`)`	Gets the item at the given index (0-based). The return type is `T`.
`set(`*index*`, `*value*`)`	Sets the entry at the given index to be the given value.
`remove(`*index*`)`	Removes the value at the given index.
`toString()`	Returns a string representation of the list.
`size()`	Returns the number of values stored in the list.

In the `Item` class, you need to construct a `String` representation of the price. This is not easy to do for several reasons, but Java provides a convenient built-in class that will do it for you. It is called `NumberFormat` and is part of the `java.text` package (so you need to `import java.text.NumberFormat`). You obtain a number formatter by calling the static method named `getCurrencyInstance()`, as in:
NumberFormat nf = NumberFormat.getCurrencyInstance(Locale.US);

You can then call the `format` method of this object, passing it the price as a `BigDecimal`, and it will return a `String` with a dollar sign and the price in dollars and cents. For example, you might say:

BigDecimal price = BigDecimal.valueOf(38.50);

String text = nf.format(price);

This would set the variable `text` to `"$38.50"`. Note that you only need *one* `NumberFormat` reference, and you can use it repeatedly. Do not create a number formatter every time you need to format a number, as that would be wasteful of memory (and would cost you points for redundancy in your implementation). Instead make a single `NumberFormat` reference that can be shared by all Item objects. An alternate acceptable way to format the output would be to use the `format()` method of the `String` class.

Your classes are to *exactly* reproduce the format and overall price shown in the two screenshots and described above. You should run the GUI and enter the individual quantities from the screenshots to verify that your classes are working correctly (and / or write a unit test that does so). Note that the values shown in the screenshots are *not* the only values your program will be tested with (and, in fact, the list of items in the screenshots is not the only list of items your classes will be tested with).

NOTE: No console output should appear when running the BookstoreMain program or when running your unit tests.

Part II (30% of the assignment grade)

Unit Testing

You are required to develop unit tests for the `Item`, `ItemOrder`, and `Cart` classes.

This part of the assignment will give you practice writing unit tests.

- You must write a individual test classes for each of the three classes under test.
- Your unit tests must be written as JUnit **5** (Jupiter) tests. The unit testing lecture slides and the Point class tests examined during class will provide explanations and examples of how to write unit tests. The lecture slides include links to additional information online. An internet search for tutorials or examples of unit testing might help you find additional useful sources of information. I am also glad to help if you are stuck.

- Your unit tests *should* pass and provide complete code coverage of the three classes under test. Use your IDE code coverage tool to check that your tests provide complete code coverage of the classes.
- Your unit tests should not produce any console output as the major way of reporting test results when run. All results of the tests should be reported by using JUnit assertions as demonstrated in class.

Writing these tests will allow you to develop these classes in stages rather than all at once. When you have confidence that your classes are working, you should combine them with the set of classes developed by the instructor to make sure that they are working properly when combined. Place your unit tests in the 'tests' package.

Hints and Suggestions:

The following is written about the **`Item`** class but is true for all three classes under test:

Each unit test should test a single method of the **`Item`** class. Each unit test ideally should have only a single JUnit assertion or test a single exception. This does not mean that the **`Item`** class and your test class will necessarily have the exact same number of methods. It may be possible to achieve complete code coverage of the **`Item`** class without writing tests for every method of the **`Item`** class (this will be true because you will use some methods of the **`Item`** class to help test other methods). In some cases, you may need to write several test methods in your test class to thoroughly test a single method of the **`Item`** class. Name each of your test methods appropriately so that the name clearly indicates which **`Item`** class method is being tested.

Do not add any public methods to the **`Item`**, **`ItemOrder`**, or **`Cart`** classes for the purpose of testing. You may need to creatively use the existing methods to achieve full coverage and gain a high level of confidence that your unit under test is correct.

For example, you may need use a combination of the **`getCartSize`** and **`toString`** to test the **`add`** method in the **`Cart`** class.

Part III (10% of the assignment grade)

Debugging Login and Registration

This part of the assignment will give you practice implementing debugging strategies and correcting code.

The Bookstore software recently added a credentialing system. The majority of the system is in place but contains some bugs. Review the following requirements and perform manual (as opposed to automated) **functional** testing to discover errors in the system. Correct any errors that you find.

Credentialing System Design

The credentialing system consists of the Java class **`CredentialingLoader`** and a text file **`files/users.txt`**. The Java class **`LoginFrame`** is the GUI for the login system. (*Do not manually edit* **`users.txt`**. *You may find it helpful to save an original copy before you begin testing. If the file becomes corrupted, you can replace it with the saved copy.*)

Credentialing System Requirements Register Requirements:

- When registering, the user must provide a username, a password, and campus (no empty fields are allowed)

- The user must type the password on the form twice and they both must match
- The username must not preexist in credentials system **Login Requirements:**
- When logging in, the user must provide a username and password (no empty fields are allowed)
- The username must preexist in credentials system
- The entered password must match the password associated with the username found in the credentialing system

Persistent Store (users.txt):

- Use a new line character as a delimiter between records
- Use the semicolon (`;`) as a delimiter between fields in a record
- Each record consists of a single user
- Each record uses the following format:
    - username;salt;hash;campus
- A line that starts with the hash (`#`) symbol is a comment
- User passwords are salted, then hashed for security, for more information:
    - https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/

Hints and Suggestions:

NOTE: The login/registration system consists of two classes, **`LoginFrame`** and **`CredentialingLoader`**. You should not alter any other provided complete classes. (**`BookstoreMain`**, **`BookStoreFrame`**, **`R`**, **`InventoryLoader`**)

NOTE: Login currently works for the username:password **test:test**.

Extra Credit

Up to 10% extra credit can be earned by completing the following challenge. When a class implements the **`Comparable<T>`** interface, it is said that this imposes a total ordering (referred to the class's natural ordering) on the objects of the class. Implement the **`Comparable<T>`** interface in the **`Item`** class. Use the value of **`Item`** objects' name first, then price second for the natural ordering. Order name alphabetically and price low to high.

Example list of **`Item`** objects using the natural ordering `(name:price)`:

(AAA:0.99) (ABC:9:99) (ABC:10.99) (XYZ:0.01) (XYZ:0.99)

There are times when developers may want to order **`Item`** objects but **NOT** by the natural ordering on the class. Instead, they would like to be able to order **`Item`** the **price**. Add a method to the **`Item`** class called **`orderByPrice`**. This method should return "*something*" that can be used by sorting algorithms to order **`Item`** objects by price. Order price low to high.

Example list of **`Item`** objects using **`orderByPrice`** `(name:price)`:

(XYZ:0.01) (XYZ:0.99) (AAA:0.99) (ABC:9:99) (ABC:10.99)

Write enough appropriate JUnit test cases to feel confident that the methods are implemented correctly. You will receive NO extra credit if the implementation is not thoroughly unit tested.

Hint: Use lambda expressions and/or method references to implement **`orderByPrice`** as a one-liner and get full credit.

NOTE: Be sure and document that you did the extra credit to ensure it is looked for and graded. Place this information in the class where you put your extra credit code. Also mention it on your executive summary.

Stylistic Guidelines:

You will be graded on program style, including the use of descriptive variable names and full Javadoc comments on each method. Note that the class skeletons provided for you have no comments of any kind; adding the comments is a critical part of the assignment. The provided `BookstoreFrame` and `BookstoreMain` classes have full Javadoc comments that would be considered acceptable in a homework submission.

Notice that the provided classes (other than the skeletons you must fill in) have minimal SpotBugs, PMD, and CheckStyle warnings. It is generally a good idea to eliminate *all* warnings before submitting your code; if you have questions about what a warning means, and a quick online search doesn't help, ask the instructor as soon as possible (preferably by posting your question on the Canvas discussion forum without posting actual code from your project).

You *must* include a header comment at the beginning of each file with some basic information, in addition to full Javadoc comments. Examples of acceptable file headers (including the class Javadoc comment) appear in the provided `BookstoreFrame` and `BookstoreMain` classes.

Submission and Grading:

Create your IntelliJ project by downloading the `hw3-bookstore.zip` file from Canvas, unzipping it, then loading it into IntelliJ.

You must submit (on Canvas) an *executive summary*, containing an "assignment overview" (1 paragraph, up to about 250 words) explaining what you understand to be the purpose and scope of the assignment, and a "technical impression" section (1-2 paragraphs, about 200-500 words) describing your experiences while carrying out the assignment.

The filename for your executive summary must be "**`executive-summary-bookstore.txt`**". Executive summaries will *only* be accepted in plain text format – other file formats (RTF, Microsoft Word, Acrobat PDF, Apple Pages) are *not* acceptable. Using an unacceptable file format for the executive summary will result in a reduction in the grade for the assignment.

Part of your program's score will come from its "external correctness." For this assignment, external correctness is measured by the output generated (correct calculations, correct behavior when values are entered, changed and deleted, correct discount behavior, *etc.*), and is determined by running automated tests on your code.

Another part of your program's score will come from its "internal correctness." Internal correctness includes meaningful and systematically assigned identifier names, proper encapsulation, avoidance of redundancy, good choices of data representation (though this is not applicable to this assignment), the use of comments on particularly complex code sections, and the inclusion of headers (as described above) on your classes. Internal correctness also includes whether your source code follows the stylistic guidelines discussed in class. This includes criteria such as the presence of Javadoc comments on *every* method and field (even private ones!), the use of variable names, spacing, indentation, and bracket placement specified in the class coding standard, and the absence of certain common coding errors that can be detected by the tools. It is therefore to your advantage to be sure the plugin tools like your code before you submit it.

For this assignment, the percentage breakdown is more or less 10% executive summary, 50% Part I, 30% Part II, and 10% Part III.