Lab 04: Intro to Object Oriented Programming

COSC 102 - Spring '24

Goal: This lab will introduce you to writing and testing your own object code with two class implementations, each with their own constructors, attributes, and methods.

1 Getting Started

On your lab's Moodle page, download the provided **`L04_starter.zip`** file. After unzipping, you will see two sub-folders: **`1_my_fraction`** and **`2_my_shopping_cart`**. Create a `Lab04` folder in your preferred location and move these folders there – it is important to create a separate directory for every assignment to keep your code organized.

2 Warmup: `MyFraction`

To begin, **only open the two `.java` files in the `1_my_fraction` folder**; you will work with the `2_my_shopping_cart` files later. Complete the tasks outlined below:

2.1 Basic Functionality

Follow the steps below to correct and complete the code that defines a `MyFraction` object, which is an abstraction of a number represented as a fraction (ex: `1/4`). Be sure to save and test after each step you complete.

1. Review the provided code. Here, skim both `MyFraction.java` and `MyFractionClient.java` to identify their responsibilities. Trace the two uncommented lines in `MyFractionClient`'s `main` method, and make a prediction as to what you think the output will be when ran.

   Next, run the code and check if the output matches your expectation.

2. Next, try **uncommenting** the `toString()` method in `MyFraction`. Predict what the output will now be of the code you ran in the previous step. Run `MyFractionClient` again and verify.

3. Uncomment the constructor in `MyFraction`. Do you expect the output of the code in `MyFactionClient` to change again? Run the client and verify.

4. Now, add a **second constructor** in `MyFraction` which accepts **two `int` arguments**: a numerator and denominator. Try instantiating/printing some `MyFraction` objects using this new constructor to verify it works.

5. We want to ensure that a `MyFraction` object will **never have a `0` value** for its **denominator** once instantiated. Implement this validation by having your constructor **throw an `ArithmeticException`** if a 0 denominator `MyFraction` were to be instantiated.

   You can throw an exception using the following syntax: **`throw new ArithmeticException("MY MESSAGE");`** You should replace `"MY MESSAGE"` with something more descriptive, and see how it manifests in your testing.

6. Now that your two argument constructor is implemented, try **commenting out** the no argument constructor again and attempt to run the previous tests in the client code. What happens? Does this make sense?

   Uncomment the no argument constructor again before continuing.

7. Create a **third constructor** which accepts a **single argument** – this constructor instantiates a `MyFraction` representing a whole number (defined by its lone argument).

8. Lastly, eliminate redundancy between your three constructors using **constructor redirection**. Remember – you can call one constructor from inside another using **`this(...)`**.

   When you're done, **two of the three** `MyFraction` constructors should only have a **single line of code** (a redirection to another constructor). You can eliminate the `print` statement that was in the provided constructor.

2.2 Instance methods

Complete the following tasks related to **instance methods** in `MyFraction`:

1. Uncomment the three lines of code under *2.2.1* in `MyFractionClient` – you will get a **syntax error**. Fix this issue **without modifying** `MyFraction.java`!
2. Uncomment the **`convertToDouble()`** method in `MyFraction`. As-is, this code will produce a **syntax error**. Correct the error and then test the method by calling it in `MyFractionClient`.
3. Implement a method **`public boolean isWholeNumber()`** in `MyFraction.java` which returns a boolean indicating if the referenced `MyFraction` is a whole number (`true`) or not (`false`). Test it in `MyFractionClient`.

2.3 Normalization

Complete the following tasks related to **normalization** in `MyFraction`:

1. Trace the four lines of code under *2.3.1* in `MyFractionClient`. Predict what the output will be, then **uncomment** these lines and run the code. Does the result match your expectation?
2. Ultimately, the four lines of code you ran in the previous step should properly indicate that `f1` and `f2` are the same. To do this, you will need to ensure that `MyFraction` objects are always in **normalized** form, meaning:

   - The numerator and denominator are stored in a most reduced form
   - If the fraction is negative, the numerator (and not denominator) is negative

   Implement the `normalize()` method in `MyFraction`, which normalizes a referenced `MyFraction`'s numerator and denominator per the criteria above – you can test it in your `MyFractionClient`. A **`gcd(...)`** helper function is provided for you; read its comments to understand its utility, and how you can use it in `normalize()`.

   Once completed, figure out where in `MyFraction.java` you should call `normalize()` to ensure that **all** `MyFraction` objects are **always in normalized form** after they're instantiated.
3. As you may have noticed, `MyFraction` objects are designed to be **immutable**; this mean once they've been instantiated, their numerator and denominator will **never change**. As such, there is **one** instance method in `MyFraction` that should be **declared `private`** (as it will never need to be called from outside the class).

   Determine which instance method this is and **change its declaration to `private`** (you may need to comment out some previous tests in `MyFractionClient`).
4. Finally, uncomment the six lines of code under *2.3.4* in `MyFractionClient`. Predict what their output will be, then run the code – you should notice something wrong!

   Review the implementation of **`createInvert()`** in `MyFraction`. Correct this method so that it obeys the class invariants defined in *2.1.5* and *2.3.2*. Once corrected, your `createInvert()` method should only be **one line of code** long (not counting the return statement).

3 MyShopCart

Navigate to the `2_my_shopping_cart` directory and open the two files, **`MyShopCart.java`** and **`MyShopCartClient.java`**.

A `MyShopCart` object represents a "shopping cart" for a single purchase on a web storefront. Our `MyShopCart`'s fields currently consist of a **customer name** and a **purchase subtotal**, though you will eventually add more.

Complete the steps outlined below. Don't forget to document your tests and remember to test after each step!

1. One of the provided files has a syntax error; fix it!
2. Next, create a constructor for your `MyShopCart` which accepts **one argument**. Since **all shopping carts start as "empty"**, what should this argument be given `MyShopCart`'s instance variables?

3. Uncomment and complete the `calcFinalTotal` method. This method returns the shopping cart's final total amount, which is the subtotal plus the sales tax (the rate of which is **`8.0%`**).

   Thus, given a subtotal of `$30.00`, this method would return **`32.4`**. It's ok if the returned value is not exactly two decimal places.

4. Add a `toString` method, which returns a `String` showing the customer's name, subtotal, and final total. For example, if a `MyShopCart` `sue` has a name `"Sue Smith"` and subtotal of 15.25, `sue.toString()` returns:

   `"Sue Smith, Subtotal: $15.25, Final Total: $16.47"`

   Again, it's ok if your `toString`'s sub/final totals shows more or less than two decimal places.

5. Your `MyShopCart` has some restrictions regarding the name associated with the purchase. If a `MyShopCart` is created with name that is **fewer than 5 characters** or **doesn't contain a space** (*i.e.: doesn't have a first and last name*), it is instead assigned the name: `"DEFAULT CUSTOMER"`. Where should you implement this?

6. Your `MyShopCart` must enforce a **minimum price** of **`$0.99`** for individual items added to a shopping cart.

   Update the **`addItemToCart`** method to return a **`boolean`** indicating whether an item is successfully added to the cart. If an adding an item is unsuccessful, this method returns `false` and **does not** modify the cart's subtotal.

7. Update **`applyCoupon`** to also return a **`boolean`** indicating whether a coupon application is successful. If a coupon would cause a cart's subtotal to be **negative** it is rejected. There's one other scenario in which a coupon would fail – think about it! Just like `addItemToCart`, an unsuccessful coupon doesn't modify its cart's subtotal.

8. Update your `MyShopCart` to add a flat **`$5`** shipping fee to its final total if the cart's subtotal is **less than `$50.00`**. This shipping fee is added to the final total.

   Specifically, discounts, taxes, and shipping fees are applied to a `MyShopCart`'s final total as follows:

   - free shipping is determined by the subtotal amount *after* coupons are applied but before taxes are added
   - taxes are calculated *after* coupons are applied but *before* any shipping charges are applied

9. Add an *instance variable* named `isTaxExempt` to `MyShopCart`. If `true`, this purchase **does not** have sales tax applied to its final total. The shipping fee is still applied (if applicable) to tax exempt carts.

   Add a *second argument* to your constructor for the tax exemption value, as well as an accessor method named **`isTaxExempt`**. Update any other necessary code to integrate this functionality into your `MyShopCart`'s logic.

   Lastly, any tax exempt cart needs **`[TAX EXEMPT]`** added to its `toString`, ex: `"[TAX EXEMPT] Jane Doe..."`

10. Add a method: **`public double calcTotalSavings()`** which returns the total money saved by the customer with this cart. The savings amount is calculated as the sum of all the applied coupon values, and also includes the `$5.00` shipping charge if the cart qualifies for free shipping.

    For example, given a `MyShopCart` that, after applying three coupons worth `$3`, `$1`, and `$2.50`, has a subtotal of `$52.54`, `calcTotalSavings()` returns **`11.5`**. You may add another instance variable to implement this feature.

    Lastly, add this savings information to be included in `MyShopCart`'s `toString()` **only if** the savings amount is greater than `$0` (otherwise, the `toString` is unchanged from the previous steps). Below is an example:

    `"Jane Doe, Subtotal: $72.25, Final Total: $78.03, (Saved: $11.51!)"`

11. Add a method: **`public boolean splitCartBill(MyShopCart other)`** which splits a `MyShopCart`'s subtotal with another cart. Specifically, the *reference* `MyShopCart` has its subtotal **reduced by half**. This same amount is then added to the subtotal of the *argument* `MyShopCart`.

    The function returns `true` if the split is successful, or `false` if not. A tax exempt cart **can only** be split with another tax exempt cart, and vice-versa. Additionally, you cannot split a cart that has had coupons applied to it (though its ok if the argument cart has coupons). Lastly, there is **no minimum subtotal** to split a cart.

12. Finally, by now there should be a few hard-coded values, *i.e.* "magic" (non-zero/empty) numbers or Strings, in your code. Identify **at least five** of these values and replace them with **`final` variables** to improve readability.

    As a reminder, the syntax to declare a final variable looks like:

    ```
    public static final [type] [VARIABLE_NAME] = [VALUE];
    ```

    Example:

    ```
    public static final int HOURS_IN_A_DAY = 24;
    ```

4 Submission

Upload your work to the **Lab 04** submission on your lab's Moodle page. You will have **four** files to submit:

- `MyShopCart.java`
- `MyShopCartClient.java`
- `MyFraction.java`
- `MyFractionClient.java`

This lab is due:

- **Tuesday, February 27th** by **5:00 PM** for lab sections **A**, **B**, and **C** (which meet on Wednesday)
- **Wednesday, February 28th** by **5:00 PM** for lab sections **D**, **E**, and **F** (which meet on Thursday)