On February 28, 2018 I was given the task to implement a checkout component. Given a list of SKU's , I was to compute the best price permutation and combination of the items in the cart.  
  
My first task was to represent the items in the cart as SKU's. The information associated with each SKU would be an ID, Color, Size, Style, Price, Promotion (including the promotion type, magnitude, start, and end date).

I set up the SKU class to have each of these descriptors be its own internal class, each with getters and setters, to better help with encapsulation. By keeping each descriptor of an SKU separate, and the SKU constructor calling several separate constructors, it helps in tracking down any given bugs that were introduced later in the project; as the stack trace would tell me specifically that the error was "Unsupported Color given" in the SKUColor constructor, instead of erroring out in the generic SKU constructor.

In addition, the separation of classes does increase maintainability if any portion of the code needs to be modified. For example, if SKUStyle needed to be more specific (As "Shirt" is probably not a useful piece of data in the real world), the engineer should know exactly where to go to update the code. Nothing in cart would need to be changed, nor anything in the SKU constructor .

Of note, there are only certain inputs allowed by the subclass descriptors. While price allows any double[[1]](#footnote-2), constructors like color or size have more specific input limits. In the case of color, the input options are those given by java.awt.color, while size is either "mens" or "womens" (note the lack of apostrophe) followed by either an M denoting a size medium, or up to 2 X's, denoting either "Extra", or "Extra Extra" and finally either an S or an L denoting "Small" and "Large" respectively. This is to say, supported sizes are XXS-XXL in both mens and womens. Where ever possible, I used equalIgnoreCase to allow for either x or X, and gray and Gray.

As far as the SKU class was concerned, nothing too difficult arose in development. Most of my time was spend modifying the auto generated getters and setters, and adding javadocs to every line.

Once the SKU class was set, the real work began in the Cart class. The cart contains the main method driver, which designated where the .csv file is saved, iterates over the cart quickly to gauge its size, and generate an array of SKU's. The array of SKU's can be abstractly visualized as the cart itself, as it's the part of code "holding" the items and their associated descriptors. Then, the method populateCart is called.

PopulateCart uses a buffered reader to read in the csv line by line, and splitting each line into a string array on each comma. Each cell of the string array represents a descriptor of the given item, in the following order: Id, Color, Size, Style, Price, Promotion Type/Magnitude, Promotion Start[[2]](#footnote-3), and Promotion End.

Once a row of the csv is SKU'ified, the for-loop increments, and reads the next line, repeating this process until the next line is null.

After all SKU's are arranged in the SKU array, it is time to apply the discounts, and total up the price. The first thing the method does is acquire the current date[[3]](#footnote-4). This current date will be used later to make sure the promotions are still valid. Several fields are also set up, including total, rebateTotal, newSale, bogo, and appliedSale[]. Total and rebateTotal are used to add up the cost of the items after the sales have been applied. The newSale Boolean flag is used to control a while loop that causes the entire cart to be reexamined, to insure that every item in the cart has been totaled. If a given loop over the cart finds a new item that had not been totaled, we stay in the while loop. The loop is only exited after we have confirmed that every item has been accounted for.

AppliedSale[] is used to make sure that a given item is not double charged, while also reducing the bigO of looping over the cart multiple times by acting as a sort of short circuit. Bogo is a Boolean flag that is used to check if we are currently in a "Buy X items, Get Y items Z% off" sale.

Once we've entered the while loop, three additional control variables are initialized to help manage bogo sales, but before we encounter any sale types, we enter a labeled for loop. I'll explain why it's labeled in a bit. This for loop will iterate over every item in the cart, starting by checking appliedSale[i] to see if that item can be safely skipped over, since it's already been totaled. It then compares the current, start and end date to see if the promotion is valid, and if it is, grabs the type/magnitude of promotion. It splits this value on the character 'x' in order to separate type into typeData[0], and magnitude into typeData[1]-[3].

Depending on what the content of typeData[0][[4]](#footnote-5) is the appropriate sale type is applied. If the sale type is none, %, or rebate, the given sales are simply applied (or not in the case of none) to that item, and appliedSale[i] and newSale are set to true. However, if the sale type is bogo, things are a tad more complicated.

Bogo follows the following logic tree.

* Are we currently in a Bogo sale?
  + If not, then set bogo to true, set bogoX to typeData[1]-1, and set bogoY to typeData[2]. Grab the id of the item we're looking at, and set bogoId to match. Add up the undiscounted price, set newSale and appliedSale, and continue.
  + If Yes, then check the value of bogoX, and make sure we're looking at the same item id
    - if X > 0, decrement X, and treat the item as undiscounted.
    - if X = 0, decrement Y, and apply the sale. If at the end of this application, Y is now zero, break back out to the label above, resetting the for loop, staying inside the while loop.

The point of using this break is to account for items that are bogo, but are not the same item as the current bogo count down. We break outside the for loop when we know we have finished the current bogo in order to return to any that we may have missed.

Once all of this is done, we print to console the total price of the cart, and any rebates available, and we are done.

1. I purposefully allowed negative values, as a possible future way to implement coupons. [↑](#footnote-ref-2)
2. Of note, the csv should always have a date given in the start and end date columns, even when there is no active promotion. Populate cart and the constructor do have contingency plans in case the column contains the string "none", but this was a relic from earlier iterations. The program will still work, but instead of saying "No sale applied", it will instead say the sale is out of date. [↑](#footnote-ref-3)
3. I'm sure there's a more secure way to get the date compared to the implemented method, but java's date works fine for a proof of concept. [↑](#footnote-ref-4)
4. the type of the promotion is denoted with a single character, n for none, r for rebate, % for %off, and b for bogo. [↑](#footnote-ref-5)