Thank you!

SYSC2004C
Object Oriented Software Development

Grade / ???

Milestone 5

Date Due: 04-14-2021
Time Due: Before 20:00 EDT
Student: Gabriel Benni K. Evensen [101119814]
Student: Ngo Huu Gia Bao [101163137]

- Inventory class
- ShoppingCart class
- ProductStockContainer class

---

1. **What is the difference between an interface and abstract class?**

   Within the context of Java programming we do not have multiple inheritance and thus we cannot use multiple interfaces; although in other programming languages you may. We may only inherit a single abstract class.

   Abstract classes do not contain the code that must be implement, rather they define common behaviours amongst child classes. Interfaces outline behaviour that implementing classes must contain.

   Interfaces may only contain abstract methods; whereas abstract classes may contain abstract classes in conjunction with others.

2. **Why we should "code against"/"program to" an interface (in relation to OOP)?**

   We should "code against" interfaces as it makes for more robust, uniform code that will work, as long as the interface remains.

   However, when coding against interfaces we need not create an interface which casts a net too wide. In other words, when coding you need a balance between deciding which interfaces you want to use; how many interfaces you want to use; and how abstract these interfaces are to be.

   Moreover, it is useful to code against interfaces as with practical work there are often many classes that carry out similar process, or are similar in some nature, thus we can simplify our coding by way of interfaces.

   Rounding off, when interfaces follow the PHU rule (I just made it up, stands for Practical, Helpful, and Useful; repetitive, I know) they are useful. When they "mirror" one or a few classes, the implementation is essentially redundant (Tor Hovland, Tor's programming blog).

3. **Change Log:**

   (a) **MILESTONE 1 CHANGELOG:- Kindly find the milestone two log below, dearest TA.**

      i. StoreManager Log:-

      For the **StoreManager** we chose to make *checkStock void* since having to manually type `System.out.println(...` each time you wish to check the stock could get tiring, very quickly. Additionally, the wording of the guideline makes it seem as this is a method the user would call when they wish to check how much of an item is left in the stock, thus this is the easier implementation for them.

      We chose to make the *processTransaction* method have a parameter consisting of a 2D array. This 2D array is then converted into a hashmap (of the same length as the array) using one of the iterator methods learned in Lab 3, and simple array operations. The variable *total* keeps track of the cumulative running total price (in an unspecified dollar currency) of the items so far. We use a simply `if` statement to check if there is adequate wares for the transaction to take place, **iff** there is, then the *total* will be incremented.

ii. Inventory Log:-

We chose to declare and initialize products in the *Inventory* constructor for now. Although, we are hoping to eventually be able to add products from the terminal, and have them dynamically added to the inventory. This implementation plan would ideally not change our StoreManager class much.

For *gettingQuantity* we decided to return an int since we are returning the number of items within the inventory. Using the knowledge gained from Lab 3, we have implemented an iterator, that looks for a specific key, within the keyset. If a match is found, we get the quantity, and return it. If there is not a match found, we simple `return -1`.

iii. HashMap:-

We decided to choose HashMap because HashMap can store different type of values which can be easily to access and get. In addition, we do not need to looping through the entire HashMap like ArrayList to find a specific value since there is the support syntax **.containsKey()**. Finally, we can add new items to the HashMap effective and quick with the syntax **.put(key,value)**

(b) **MILESTONE 2 CHANGELOG:-**

i. StoreManager Log:-

We chose to add the method **assignNewCartID** which is responsible for the designating each new cart with a distinct ID. The way by which we created distinct ID's is described below.

We chose to add **removeCartInventory** which is responsible for removing items from a shopping cart; of a designated quantity.

We implemented the **emptyCustomerCart**. This method will be called inside the quit method of the **StoreView**. For the situation that the customer quit the store without checking out and move to other store. It will loop through the customerCart HashMap which is the customer's cart, the method will remove all of the items and put them back to the **Inventory**.

We **moved the main** to our StoreView.java class as is outlined in the milestone two handout. It does also make sense to this do this as StoreView might be considered our class of the greatest scope.

ii. Inventory Log:-

We chose to add **getAvailableID** method when we need a full Set of Integers of our keys in the infoProduct attribute.

iii. ShoppingCart Log:-

We chose to have an Integer ID for each shopping cart instance. We initialized this variable to zero; upon calling the constructor it is incremented.

We chose to implement **addCustomerProduct** method which takes two Integers; an ID, an amount and an Inventory object. This method is responsible for handling a customers request to add items to their shopping cart. This method must check if the amount that the customer wishes to add is greater than the current available amount; if so an error is printed.

We chose to implement **removeCustomerPrducts** method which takes two Integers; an ID and an amount. This method is responsible for handling a customers request to remove items from their shopping cart. This method must check if the amount that the customer wishes to remove is lesser than the current available amount; if so an error is printed.

iv. StoreView Log:-

We chose to implement **helpDisplay** such as to help with displaying the different options when a user tupes "help".

We chose to implement **browseDisplay**, which serves as the GUI when a user makes the decision to browse the different products. It lists all products; their ID's; their respective names; and their price per unit.

We chose to implement **addDisplay**, which serves as the GUI when a user makes the decision to

purchase an item (i.e., add an item to their shopping cart). It lists all the available products, and their respective information. It also prompts the user to enter the ID of the item which they wish to purchase, followed by the amount they which to purchase.

We chose to implement **removeDisplay**, which serves as the GUI when a user makes the decision to return an item (i.e., remove an item from their shopping cart). It prompts the user to enter the ID of the item they wish to return; and the amount they wish to return.

We chose to implement **displayUI** to navigate, according to the user's input, through our different methods, and store.

4. **MILESTONE 3 CHANGELOG:-**

(a) ShoppingCart Log:-

We changed the **addCustomerProduct** method which now takes two Integers; an ID and an amount. The method used to take two Integers; an ID, an amount and an Inventory object. The purpose is the Shopping Cart and the Inventory class will be independent and the managing of the Store Manager class will be thoroughly. The function of the method is still the same. This method is responsible for handling a customers request to add items to their shopping cart. This method must check if the amount that the customer wishes to add is greater than the current available amount; if so an error is printed.

(b) StoreView Log:-

We added the handling error in the **addDisplay** method. The method first will prompt the user to enter the ID of the item which they wish to purchase, followed by the amount they want to purchase. If the user enters the invalid ID or the amount (ex: String), the method will catch that error and return a message to the users.

We added the handling error in the **removeDisplay** method. The method first will prompt the user to enter the ID of the item which they wish to remove from their cart, followed by the amount they want to remove. If the user enters the invalid ID or the amount (ex: String), the method will catch that error and return a message to the users.

We added one more if statement in the **displayUI** to check if the user enters invalid command, the support message will be printed out.

(c) Testing Methods

i. **Inventory class**
We have three testing methods for this class.

- **testGetAvailableID** which will check if the **getAvailableID** method returns the available ID of the products.
- **testGettingQuantity** will check the if **gettingQuantity** method returns the correct quantity of the products when the user input the ID of the product.
- **testRemovingQuantity** will check if the **removingQuantity** method removes the quantity of the product with the given ID and the amount of the product.

i. **StoreManager class**
We have five testing methods for this class.

- **testAssignNewCartID** which will check if the **assignNewCartID** method increment the cartID by one and store it as a key, and the Shooping Cart object as a value inside the HashMap.
- **testRemoveCartInventory** will check the if **removeCartInventory** method adds the wanted amount to the customer cart and remove the correspond amount from the inventory.

- **testAddCartInventory** will check if the **addCartInventory** method removes the quantity of the product with the given ID and the amount from the customer cart and adds them back to the inventory.

- **testProcessTransaction** will compare the expected string with the returning actual string of the **processTransaction** method.

- **testEmptyCustomerCart** will check if the **emptyCustomerCart** method will remove all the items inside the customer cart and put them back to the inventory.

5. **MILESTONE 4 CHANGELOG:-**
We decided to remove the **helpDisplay, browseDisplay, addDisplay, removeDisplay and displayUI** methods as we implemented the GUI for these methods, in this milestone. When the programme executes, the implemented GUI will display such that the user-interface interaction will be much more intuitive. The users now need not enter the command in the console, they may just press the buttons corresponding to each action.

Furthermore we decided to remove the ability to change storeviews (as was required in Milestone 2) as this would add much programming overhead. We realised that if the functionality worked for one storeview it should work for multiple; and typically when shopping (i.e., Amazon, ebay, etc.) there is not an option to change the storeview, we simply did not see the necessity.

In addition, the method **emptyCustomerCart** inside the **StoreManager** and its test method are removed. **emptyCustomerCart** handle the situation when the customer quit the store without checking out and move to other store. They are removed because we only have 1 store view and the user can only terminate the program to exit the GUI in this milestone.

Our GUI has 4 layouts in total (ordered from outside to inside):-
mainPanel (BorderLayout) ⇒ bodyPanel (GridLayout) ⇒ storePanel, and cartPanel (BoxLayout) ⇒ itemInStore, and itemInCart (GridBagLayout)

(a) **mainPanel** contains
   i. headerPanel:- "The Milky Way"
   ii. bodyPanel:- See below
   iii. footer Panel:- "Checkout" Button

(b) **bodyPanel** contains
   i. storePanel:- See below
   ii. cartPanel:- See below

(c) **storePanel** contains
   i. storePanelHeader:- "Store"
   ii. storePanelBody:-
      A. itemInStorePanel: image, product information, slider, and add product button
      B. Vertical scrollbar

(d) **cartPanel** contains
   i. cartPanelHeader:- "Cart"
   ii. cartPanelBody:-
      A. itemInCartPanel: image, product information, slider, and remove product button
      B. Vertical scrollbar

StoreView class
We have added 7 methods for this class.

- **displayProduct(int id)**: This method will display the product information, in the store panel of the Inventory class and put it to the new JLabel

- **displayCartItem(int id)**: This method will display the product information in the cart panel of the ShoppingCart class and put it to the new JLabel

- **imageMapping(int id)**: This method will display the image corresponding to the id of the product with the width 150 and height 150 and add the image to the new JLabel

- **productDisplay()**: This method will loop through all the products in the Inventory and create the GridBagLayout each time for each product. This method called the imageMapping and the displayProduct method to give the information of the product. The Add to cart button will call the removeCartInventory from the StoreManger inside its actionListener method, so that the users can add the product to their cart

- **cartDisplay()**: This method will loop through all the products in the ShoppingCart and create the GridBagLayout each time for each product. This method called the imageMapping and the displayProduct method to give the information of the product. The Remove from cart button will call the addCartInventory from the StoreManger inside its actionListener method, so that the users can remove the product from their cart; to the store.

- **footerDisplay()**: This method will create the check out button at the footerPanel. The checkout button will call the processTransaction method from the StoreManger class which will display the receipt message. After the OK button inside the receipt message is clicked, the program will exit

- **displayGUI()**. This method will add all the panels to the frame and display the GUI

MILESTONE 5 CHANGELOG:-

For this milestone we had to remove a lot of redundant method. We also had to do a complete re-haul on many of our methods; below I have outlined these exact changes:-

1. `public int getProductQuantity(Product product)` in Inventory:- We changed the parameter here from the ID of the product to the actual Product itself. We did this in accordance with the requirements of this milestone, needing a Product passed in.

2. `public Set<Product> getAvailableProduct()` in Inventory:- We changed the return value from a `Set<Product>` to a `HashSet<Product>`. We did this as we were previously returning an `ArrayList<>` which does not work.

3. `ProductStockContainer.java` is an interface that encompasses `Inventory.java` and `ShoppingCart.java`. We were careful to "code against" this interface by using methods which belong to both of the classes, while still not being too specific.

4. `public int getProductQuantityCart(Product product, int cartID)` in StoreManager:- We had to change the parameter to a `Product` as per the guidelines of the milestone.

5. `private JLabel displayProduct(Product product)` in StoreView:- We changed the parameters of this method once more as the guidelines of milestone 5 tell us we must pass a product.

6. `private JLabel displayCartItem(Product product)` in Storeview:- Similarly, we changed the parameters to conform with the milestone guidelines.