Thank you!

# SYSC2004C
# Object Oriented Software Development

Grade / ???

## Milestone 2

**Date Due: 08-03-2021**

**Time Due: Before 20:00 EDT**

**Student: Gabriel Benni K. Evensen [101119814]**

**Student: Ngo Huu Gia Bao [101163137]**

---

- The Inventory class
- The StoreManager class
- The StoreView class
- The ShoppingCart class

---

1. **What kind of relationship is the StoreView and StoreManager relationship? Explain.**

   The relationship between **StoreView** class and **StoreManager** is that of *composition*. The **StoreView** class has a **StoreManager** object as a reference; thus, the **StoreView** can use, and access the **StoreManager** class. We decided to choose *composition* over the *inheritance* as *composition* is more flexible when implementing. This means that the changing is typically not problem-prone. Moreover, the complexity of the algorithm is greater when using *inheritance*.

2. **Due to their behavioral similarity, would it make sense to have ShoppingCart extend Inventory, or the other way around? Why or why not?**

   Having **ShoppingCart** *extends* **Inventory** is what seems to make the most sense, at this point, and from our understanding. Firstly; both **ShoppingCart**, and **Inventory** are similar in their behaviours wherein which they have to use the add quantity, and remove quantity methods. When using *inheritance*, we need not repeat the code, and we might avoid the redundant code in the subclass. Secondly; if ShoppingCart *inherits* the **Inventory**, we need not test the add quantity and remove quantity methods because they have been tested previously.

   We may also make the **Inventory** object inside the **ShoppingCart** class such that the **ShoppingCart** class might access all the methods of the **Inventory** class. However, implementing it this way can make the code more complex to recognise, as the "**.**" properties may be more ambiguous.

3. **What are some reasons you can think of for why composition might be preferable over inheritance? These do not have to be Java-specific.**

   *Composition* is typically preferred to *inheritance* as **Java does not support multiple inheritance**. Furthermore, one extend class might not be enough for the subclass. Secondly; *inheritance* is not always straightforward to implement. For example, having the **Bird** class as the superclass, and letting the subclasses be **Robin**, and **Penguin**. By *inheritance*, both subclasses might inherit all of the characteristics from the superclass; including **flying**, but **Penguin** class may not fly. Finally, *inheritance* helps to make code more unambiguous; the more we factor out, the more the complex the function may be. The aforementioned **flying** problem may be solved by implementing a **Birds can fly** superclass of **Robin**, and a **Birds can't fly** class for the **Penguin** class and the superclass of them will be **Birds**. Although this implementation may have solved the problem, the complexity of this algorithm is too great.

4. **What are some reasons you can think of for why inheritance might be preferable over composition? These do not have to be Java-specific.**

   *Inheritance* is typically preferable to *composition* as the "is-a" relationship between a child and a parent class. For example:-

A window **is a** car's component
A car **is a** vehicle

*Inheritance* allows for the reuse the code while *composition* does not. Reusing code leads to lesser ambiguity as the code need not be repeated; i.e., lowering redundancy. Moreover, we need not test those codes in the subclass again since they have already been implemented and tested in the super class. In other words; if an issue were to occur, the problem would not lay with the inherited code, but rather with the new code.

5. **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 **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 and an amount. This method is responsible for handling a customers request to add items to their shopping cart. This method must check if the amount that tne 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.