ITCS 209 – Object Oriented Programming
Project 02: Grocery Store Data Management

Update Logs:
2021-03-29 Initial Version (v1.0)
2021-04-07 Updated TestCaseTask5.java & Changed parameter from VOID to VOIDED in p.7 (v1.1)

Due: Friday, April 30, 2021 11:55PM

Learning Objectives:

On the course of implementing this programming project, you will learn some of the basic concepts of object oriented programming and how to apply them to practical, real world programming applications. Specifically, upon accomplishing this project, we expect you to be able to:

1. Instantiate objects and use them.
2. Implement classes given from interfaces.
3. Learn how and when to use List, Set and Map data structures.
4. Read/Write simple text files.
5. Learn how to use regular expressions and simple string manipulation techniques.
6. Use basic algorithm such as sorting with the object
7. Get a glimpse of how OOP can be applied to real-world problems.
8. Enjoy coding with Java.

Introduction:

In this project, you will create a program to handle order in the grocery store. The store will have a stock containing several types of items. A customer can place an order either in-store or online. For any online order, there is a shipping fee based on the distance. Customers can pay for their orders using either cash, credit card, or e-wallet. For the credit card and e-wallet payment, customers have to provide some specific information to authorize the payment. These stock items, customers' information, and orders can be stored and retrieved from the log files so that the owner can analyze the data later. For example, filtering the orders that paid by e-wallet, sorting the orders based on their grand total, grouping the total paid amount by payment method, finding the best seller items and etc. This program is similar to simple version the database management system in the real world. ^_^

Set of classes and test cases are provided for you. You may find some of the existing implementations useful. Feel free to modify these files by adding variables and methods to facilitate your algorithm. However, you must not modify code in the "`DO NOT MODIFY`" zones. Since there are several tasks in this project, you should try to incrementally test your program with the provided test cases in order (Testcase1, Testcase2, and so on). The class diagram with some of the methods that you have to implement is shown in Figure 1. The explanation about the methods is provided as comments in all the java files.

Grocery Store Class Design

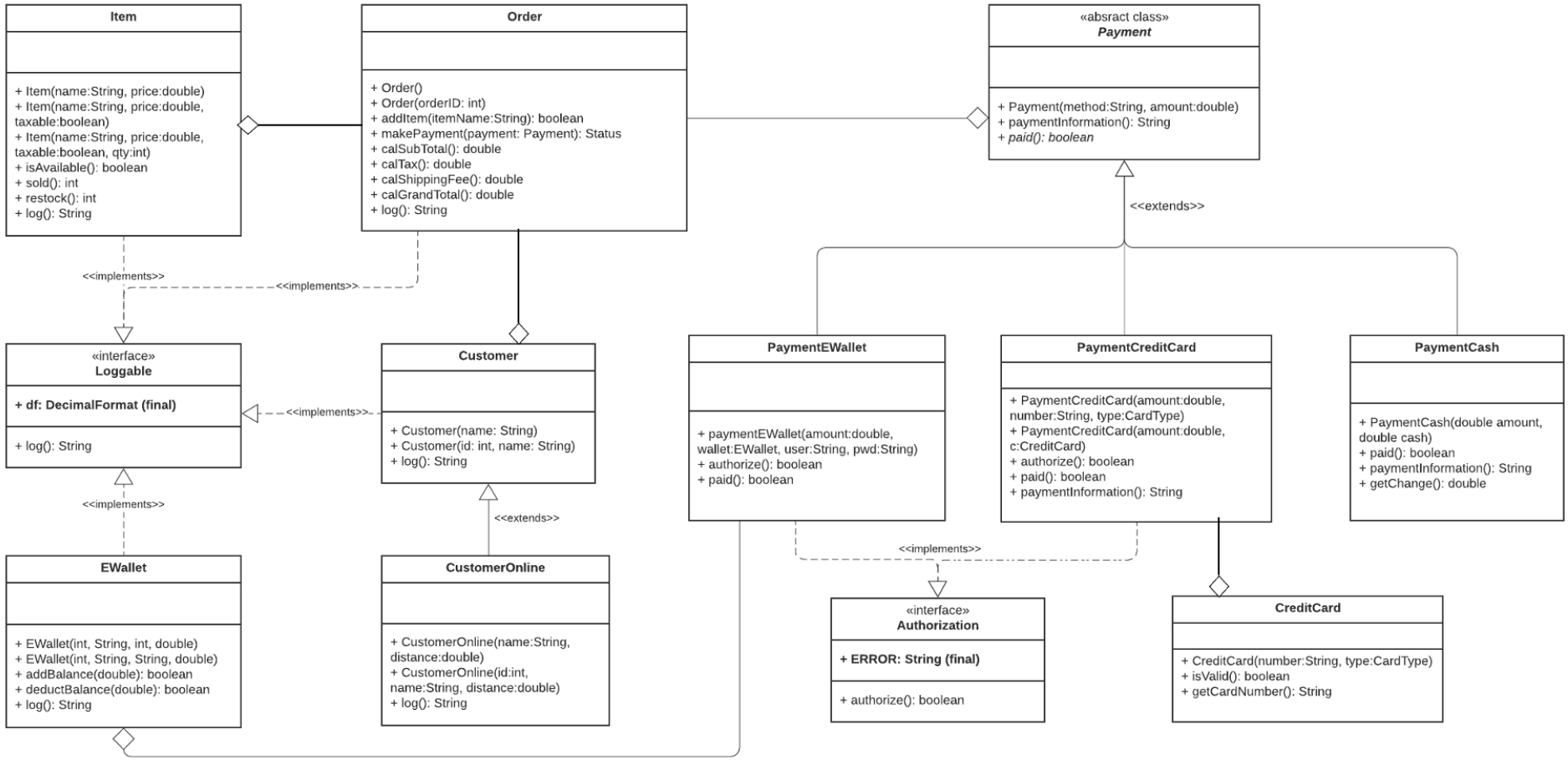


Figure 1: Grocery Store Class Diagram

(note: the completed version of the diagram is provided in the separated file)

Task 1: Implement Item, Customer, EWallet, CreditCard and Payment Classes

Item

The `Item` class represents different types of items selling in the store e.g., milk, coffee, and bread. Each item must have name, price, and quantity in stock. Also, some items need to include sell tax, so you need an indicator to identify the taxable item. The explanation of all methods is provided as the comment in the **item.java** file.

Customer

Each customer must have name and ID as shown in the `Customer` class. The ID can be either auto generated (using running ID static filed) or explicitly identify in the constructor method. For an online customer, he/she has to further provide the shipping distance to his/her home. As shown in the diagram, the `CustomerOnline` class extends the `Customer` class. Additional variables and methods are needed.

EWallet

Customers can be associated with `EWallet` accounts. So that they can pay for their orders using the electronic wallet payment method. Each wallet must have customer's ID, username, (encoded) password, and current balance. For the security purpose, the password cannot be stored directly as a plain text. The `hashCode()` method must be used to encode the plaint text to integer value.

For example, `"myP@ssW0rd".hashCode()` will get the value of -1840995161

CreditCard

A credit card information includes card number and card type. Different card type has different constraint on the card number as follow

Type	Card Number		Example of Valid Card Number	Formatted Card Number
	Length	Start with		
VISA	16	4	4234567890123456	#### #### #### ####
JCB	16	3528 to 3589	3550567890123456	#### #### #### ####
MASTERCARD	16	51 or 52	5234567890123456	#### #### #### ####
AMERICANEXPRESS	15	34 or 37	371256789012345	#### ###### #####

Payment

The store accepts variety of payment methods including *cash*, *credit card*, and *electronic wallet.* Since each payment method has different way to handle the payment transaction, the method "`paid()`" in the `Payment` *abstract class* cannot be implemented yet and has to defined as an *abstract method* as shown in Figure 1. Three subclasses named `PaymentCash`, `PaymentEWallet`, and `PaymentCreditCard` have to extend Payment and implement that unimplemented method `paid()`. Unlike cash payment, the electronic wallet and credit card payment require a customer to authorize the payment transaction. So both classes have to implement the `Authorization` interface and implement the unimplemented abstract method named `authorize()`. The explanation of all methods is provided as the comment in the related source code files.

NOTE: Make sure that each payment class extents correct class and implements correct interface. You do not have to implement the Loggable interface yet. This will be explained later in Task 5.

Check your code with the given TestCastTask1.java

Task 2: DataManagement.java -> Retrieve Objects from Log Files

-- Note that you have to finish Task 1 before working on Task 2 --

List of customers, items, and wallets can be stored and retrieved via the logs file. The `DataManagement` class has `customerData`, `stockData`, and `walletData` variables as HashMap to store list of customers, stock of items, and list of wallets respectively. In this task, the log files **(stocks.txt, customers.txt, and wallets.txt)** are provided in the **testcase_init** folder. You have to write methods in **DataManagement.java** to read each text file, re-create list of objects and store them in the HashMap collection. Each line with the valid format can be recreated into an object. The line with invalid format will be skipped.

Class	Valid Line Format	Valid Format Examples	Invalid Format Examples
Customer	custID,name	1,Andrew 2,Sarah	3,3,Ann Bob,4
CustomerOnline	custID,name,distance	5,Elle,50.00 6,Noel,1000.00	7,100.00,Potter 9,"Elsa",10.0
Item	name,price,taxable,qty	bread,40.00,true,10 cocoa,320.00,true,2147483647	bread, ,false,-10 milk,false,10.00
EWallet	custID,user,pwd,balance	1,AAA,201489,2000.00 2,B23B,-148889,2000.00	3,ABC,plain_pwd,1000.00 4,XYZ,9348934

As shown in the figure below, the customers.txt contains five lines of text to represent list of customers. However, only two out of five lines are valid. The first line "1,Andrew" represents a general customer whose ID is 1 and name is Andrew. The forth line "5,Elle,50.00" represents an online customer whose ID is 5, name is Elle, and the distance is 50.00.

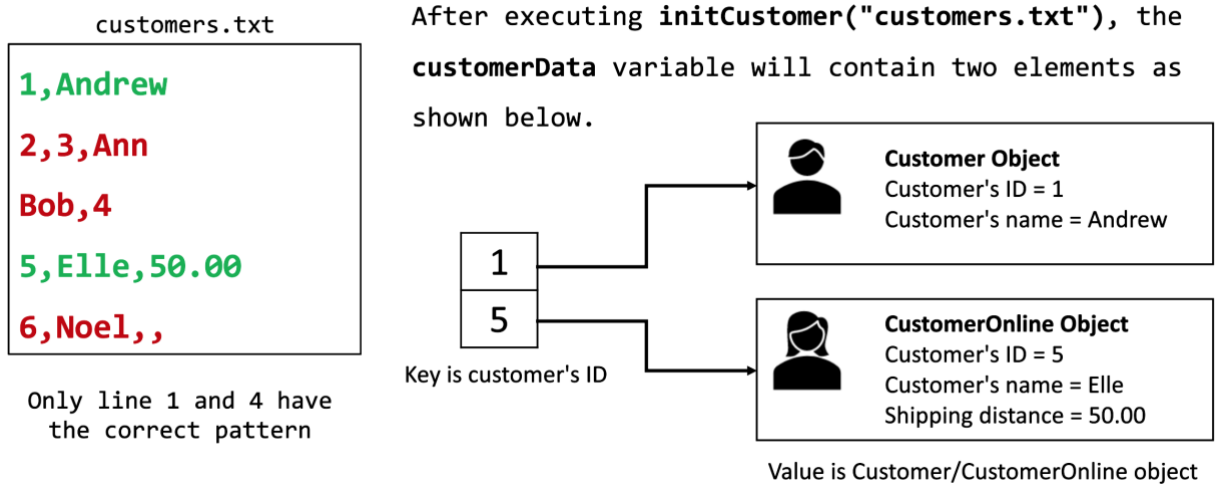


Figure 2: customerData after executing initCustomer("customers.txt") method

Check your methods with TestCaseTask2.java

Task 3: Implement Order Class

-- Note that you have to *only* finish Task 1 before working on Task 3 --

A customer can make an order by adding multiple items into his/her order and finally make a payment. The customer must already exist in the list of customers (found in the `customerData` collection). If the item is available, the quantity of the item in the stock will be deducted by one. If the payment is invalid (such as insufficient balance in the e-wallet, or unauthorized credit card), the item will be restocked. There are only three types of payment method CASH, CARD, or EWALLET. If a customer wants to pay with an e-wallet, that wallet must already exist in the list of wallets (found in the `walletData` collection) as well. The subtotal of the order is the summation of all items in the list. Some items are taxable, so the additional tax has to be calculated. In addition, orders made by online customers must be shipped to them. As a result, the shipping fee is added into the grand total of the order. The tax rate and the shipping rate are declared as the static final variable TAX_RATE and SHIPPING_RATE.

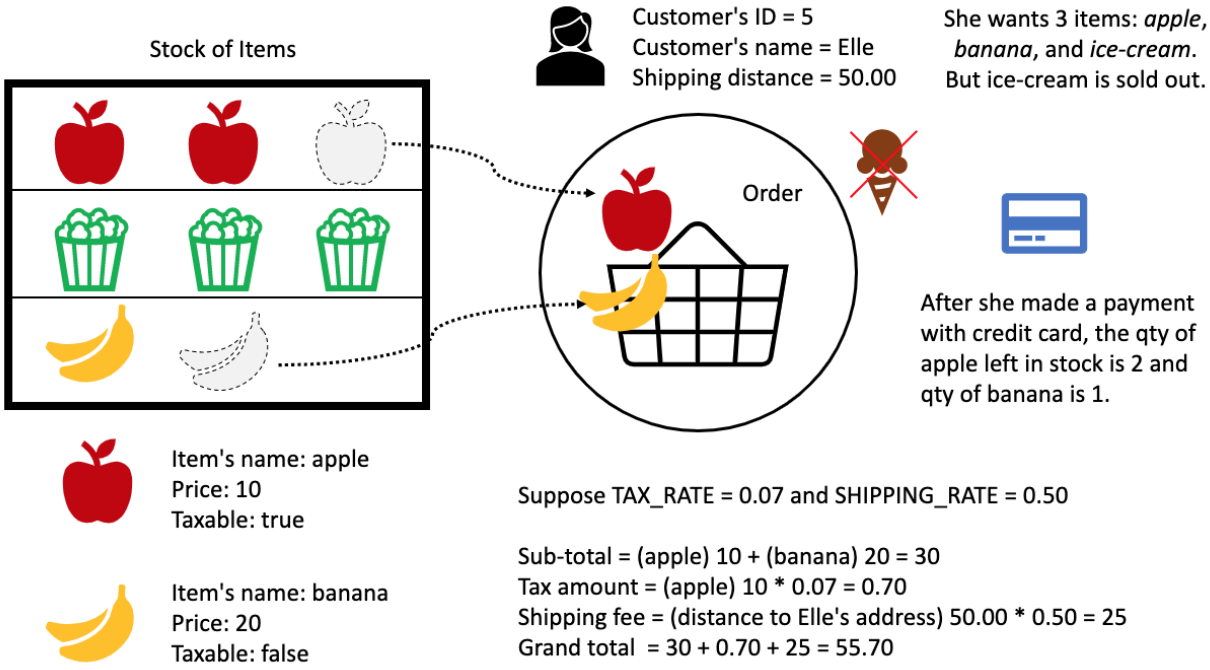


Figure 3: Example of an order made by a customer [customer's id = 5] containing apple and banana item paid by credit card

Check your methods with TestCaseTask3.java

Task 4: DataManagement.java -> Retrieve "Order" Objects from Log Files

-- Note that you have to finish all Task 1, 2 and 3 before working on Task 4 --

Similar to Task 2, you have to implement **`initOrder`** method in **`DataManagement.java`** to read **orders.txt** in the **testcase_init** folder, re-create list of orders and store them in the `orderData` HashMap collection. Since this process is the recreation of the past orders which has already sold to the customers, the actual stock will not be affected by these orders. When items are added into the list, the quantity of items in the stock will not be deducted.

Each line with the valid order's format can be recreated into an object. The line with invalid format will be ignored. The valid format is

`orderID,custID,item1|item2|…|itemN,paymentStatus::paymentMethod`

**paymentStatus can be either VOID, PENDING, or PAID*
**paymentMethod can be either UNKNOWN, CASH, CARD, or EWALLET*

Example of Line	Valid?	Explanation
3,2,coffee|bread|apple,VOIDED::CASH	Yes	orderID=3, custID=2, items=<coffee,bread,apple>, status=VOID, and method=CASH
4,5,ice cream,PENDING::UNKNOWN	Yes	orderID=4, custID=5, items=<ice cream>, status=PENDING, and method=UNKNOWN
5,5,honey|popcorn,PAID::EWALLET	Yes	orderID=5, custID=5, items=<honey,popcorn>, status=PAID, and method=EWALLET
6, , banana|popcorn,PENDING::UNKNOWN	No	Missing custID
7,1,cheese|popcorn,UNKNOWN::CASH	No	Incorrect payment status
8,1,"ice cream"|"coffee",VOIDED::TRANSFER	No	Incorrect items format and payment method

Check your methods with TestCaseTask4.java

Task 5: Implements Interface and Write to Log Files

-- Note that you have to finish only Task 1 and 3 before working on Task 5 --

As shown in Figure 1, four classes have to implements `Loggable` interface including Customer, Item, EWallet, and Order. For each class, the `log` method must be implemented to return String in the valid format as explained in Task 2 and 4. This log method will be used in the DataManagement class to write the data into the text file. To complete this task, you have to also implement storeCustomers, storeItems, storeEWallets, and storeOrders methods in the DataManagement class. *Don't try to do everything from scratch!* The provided methods in this class are useful in completing this task.

Check your methods with TestCaseTask5.java

Task 6: Analyzing Data

-- Note that you have to finish all Task 1 and 3 before working on Task 6 --

The owner wants to analyze the data in the system. You will have to implement some methods to facilitate them; for example, finding the total payment of orders that paid using credit card, showing the total payment of all orders group by the payment method, or sorting the orders based on their grand total payment. Here is some example of the data and results.

Suppose there are these data in the customerData, stockData, and orderData

Customers	Items	Orders
1,Andrew 2,Elle,50.00	apple,10.00,true,100 banana,20.00,false,200 popcorn,30.00,false,200 honey,40.00,false,200 ice cream,50.00,false,200 bread,100.00,true,200	1,1,apple|banana,PAID::CARD 2,1,apple|banana,PAID::CASH 3,1,apple|bread|apple,VOIDED::CASH 4,2,ice cream|bread,PAID::CASH 5,2,honey|popcorn,PAID::EWALLET

From those orders, we can spit their properties as follow:

OrdersID	Status	Method	subtotal	tax	shipping	grand total
1	PAID	CARD	30.00	0.70	0.00	30.70
2	PAID	CASH	30.00	0.70	0.00	30.70
3	VOIDED	CASH	120.00	8.40	0.00	128.40
4	PAID	CASH	150.00	7.00	25.00	182.00
5	PAID	EWALLET	70.00	0.00	25.00	95.00

Example usages of the methods in this task:

1) Summation of the sub total of the orders paid with cash
```
filterSubTotal(Status.PAID, "CASH");     // 30.00 + 150.00 = 180.00
```
2) Summation of the grand total of the voided orders paid with cash
```
filterGrandTotal(Status.VOIDED, "CASH");          // 128.40
```
3) Show the summation of the grand total of paid order and group by payment method
```
groupGrandTotalByPaymentMethod();
```

Payment Method (Key)	Total of Grand Total (Value)
"CASH"	212.70
"CARD",	30.70
"EWALLET"	95.00

4) Sort the order based on the grand total of “paid” order in either ascending or descending order and return in the ArrayList of Order
```
sortPaidGrandTotal(true);
```

Index	Order ID	Grand Total
0	1	30.70
1	2	30.70
2	5	95.00
3	4	182.00

Check your methods with TestCaseTask6.java

Flexibility Policies (similar to Project 01):

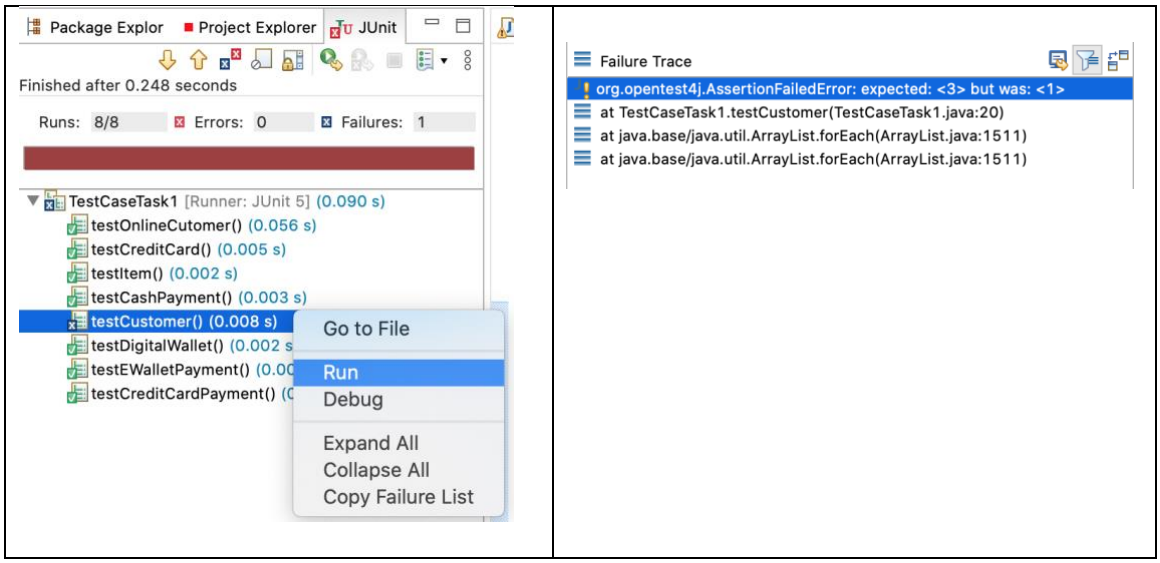
We understand that there are many OOP and Java features that you would like to explore and use to present your creativity. However, some guidelines must be set to allow fair evaluation while leaving room for flexibility to explore many wonderful Java-based OOP features.

1. Additional variables/methods: You are free to implement your own additional class variables and methods that facilitate the implementation of your algorithms. However, you must not modify the code inside the "`DO NOT MODIFY`" zones.
2. Additional classes: You are also free to add additional classes to facilitate your implementation. Also include additional class implementation files (i.e. YourOwnClass.java) with your submission.
3. External libraries: You are allowed to use third-party Java libraries. However, you have to explain about the libraries in your report and send the jar file with your submission
4. Result differentiation: We understand that when it comes to actual implementation, small details can slightly differ. Therefore, when we grade your project, a small deviation of ± 10% is OK.

Suggestions:

1. [**Understand the existing code first**] It helps to understand the provided implementation before. So, you know what is available for you to use, and what needs to be implemented.
2. [**Start early**] Don't wait until the last weekend to start working on this project. Start early, so you have enough time to cope with unforeseen situations. Plus, it can take some time to completely understand the project's specs.
3. [**Incremental tests**] The details of this project can be overwhelm for you during the course of implementation. Our suggestion is to spend some time understanding the requirements from the specs. Then, incrementally implement and test one method/part at a time, to see if it behaves as expected. It is discouraged to implement everything up and test it all at once.

   With the Junit test, you can test each test case individually, by right click at the method and select run. If there is an error from the run, the result will be shown in the Failure Trace. You can click on the message to jump to that failed statement. (If you want you may create your own testcase as well.)



Submission Instruction:

It is important that you follow the submission instructions. Failing to do so may result in a deduction and/or delay of your scores.

1. Create a folder and name it P02_<your *Student ID*>. Let's call it the submission folder.
2. Put the following files into the submission folder (*note that the filenames must be exactly the same as shown here. Otherwise, the autograder will not recognize the files.):
   - `CreditCard.java`
   - `Customer.java`
   - `CustomerOnline.java`
   - `DataManagement.java`
   - `EWallet.java`
   - `Item.java`
   - `Order.java`
   - `PaymentCash.java`
   - `PaymentCreditCard.java`
   - `PaymentEWallet.java`
3. **(optional)** For the challenge bonus, you have to submit at most two pages report explaining what you do and how to run/test your implementation.
4. Zip the folder. (E.x., `P02_63881234.zip`). Make sure that your ID in the filename is correct.
5. Submit the zip file on MyCourses before the deadline.
6. Redownload the submission and recheck the ID in the submission package, and make sure it is what you wanted to submit. MyCourses can be ridiculous if you submit multiple times.

Note: Late submission will suffer a penalty of 20% deduction of the actual scores for each late day. You can keep resubmitting your solutions, but the only latest version will be graded.

Coding Style Guideline:

It is important that you strictly follow this coding style guideline to help us with the grading and for your own benefit when working in groups in future courses. Failing to follow these guidelines may result in a deduction of your project scores.

1. Write your name, student ID, and section (in commented lines) on top of every ***Java*** code file that you submit.
2. Comment your code, especially where you believe other people will have trouble understanding, such as each variable's purposes, loop, and a chunk of code. If you implement a new method, make sure to put an overview comment at the beginning of each method that explains 1) Objective, 2) Inputs (if any), and 3) Output (if any).
3. Do not put your code in new packages. Put everything in the default package. (i.e. there should not be a package declaration on top of each java file).

Grading Criteria:

Some tasks depend on the previous tasks. So, make sure that you read the project description carefully and work on each task in an appropriate manner.

1)	Follow the submission and coding style guideline	5%
2)	On-time Submission	5%
3)	Task 1: Implements classes Item, Customer, CustomerOnline, EWallet, CreditCard, Payment and related classes	20%
4)	Task 2: DataManagement.java -> initCustomer, initStock, initWallet	20%
5)	Task 3: Implement Order class	10%
6)	Task 4: DataManagement.java -> initOrder	10%
7)	Task 5: Implement Loggable Interface and write to log files	10%
8)	Task 6: Analyzing Data	20%
	Total Score	**100%**

[Optional] Bonus for creativity (Max 20%)

You are encouraged to implement your own cool feature for this project (~be creative~) such as creating an interface to accept the input from a user and create the order, making new types of payment method, new functions to analyze the data such as finding top 10 best-selling items, or finding which items are usually bought together using *Market Basket Analysis*[1]. To qualify for the bonus, you also have to submit at most two pages report describing your new feature.

Bug Report:

Though not likely, it is possible that our solutions may contain bugs. In this context, a bug is not an insect, but an error in the solution code that we implemented to generate the test cases. Hence, if you believe that your implementation is correct, yet yields different results, please contact us immediately so proper actions can be taken.

Need help with the project?

If you have questions about the project, please first ask them on the class' general channel on Teams, so that other students with similar questions can benefit from the discussions. The TAs can also answer some of the questions and help to debug trivial errors. If you still have questions or concerns, please make an appointment with one of the instructors. ***We do not debug your code via email***. If you need help with debugging your code, please come see us.

Academic Integrity (Very Important)

Do not get bored about these warnings yet. But please, please ***do your own work***. Your survival in the subsequent courses and the ability to get desirable jobs (once you graduate) heavily depend on the skills that you harvest in this course. Though students are allowed and encouraged to discuss ideas with others, the actual solutions must be originated and written by themselves. Collaboration in writing solutions is not allowed, as it would be unfair to other students. Students who know how to obtain the solutions are encouraged to help others by guiding them and teaching them the core material needed to complete the project, rather than giving away the solutions. ***You can't keep helping your friends forever, so you would do them a favor by allowing them to be better problem solvers and life-long learners.*** Your code will be compared with other students' (both current and previous course takers) and online sources using state-of-the-art source-code similarity detection algorithms which have been proven to be quite accurate. If you are caught cheating, serious actions will be taken, and heavy penalties will be bestowed on all involved parties, including inappropriate help givers, receivers, and middlemen.

[1] https://www.kdnuggets.com/2019/12/market-basket-analysis.html