

Name: Derek Aguirre

Date: 9/27/2020

CS 3331 – Advanced Object-Oriented Programming – Fall 2020

Instructor: Daniel Mejia

Assignment: Programming Assignment 2

I confirm that the work of this assignment is completely my own. By turning in this assignment, I declare that I did not receive unauthorized assistance. Moreover, all deliverables including, but not limited to the source code, lab report and output files were written and produced by me alone.

1. Program Explanation

In this programming assignment the goal is to expand on the first programming assignment and add more functionality to the program. This assignment requires creating and extending abstract classes, reading and storing a csv file with user accounts, handling deposits, withdraws, money transfers, paying other individuals, balance inquiries, generating an updated balance sheet of all users as a csv, and logging all transactions between users. Most of my time was spent trying to figure out how the classes interacted with each other. The task was difficult to tackle at first, but once I worked finished the class functionality, everything started to make more sense.

2. What did I learn?

I learned a great deal of information about programming in this assignment, I would say more than I ever had up until now. Learning it in the classroom is completely different from implementing the concepts into my program. While I am mostly pleased with my assignment results, I feel that there is more that can be improved than not.

One of the other things I could improve on is my exception handling and the logic used for handling the user choices. The logic of the user choices is mostly within while loops and if the customer or bank manager enters in a choice incorrectly within the main directory of the code, it ends the code, otherwise, the code works as intended.

3. Solution Design

I'll start off with the logic for my classes. I have an abstract Person class that holds all the information that we are given that pertains to a person. This class has firstName, lastName, dateOfBirth, address, and phoneNumber all saved as strings. I have only one other abstract class which is of Account, it holds an int accountNumber and a double startingBalance. I also have three subclasses of Account that represent Checking, Savings, and Credit accounts. All of the respective classes extend Account's attributes and methods with the exception of Credit having a different deposit method due to its unique qualities. Now that I had all of these classes created, it was time to make a class for a customer. This is where most of my problems stemmed from when it came to the logic of the class. I created a Customer class which extended Person so now it has access to all of the information about a person. Once I inherited from Person, I realized that I still needed their account information because a customer needs to have access to their account information. For a long while I was stumped about how to access the information from all the accounts until I realized that I can create instances of all the accounts in the Customer class. I am not sure if this was the intended way I was supposed to implement the association between these classes, but it worked for me.

Next is the logic used in the RunBank class, in part one of the programming assignment, most of the functionality ran within the main method. This time however, most of the code has been migrated to methods to help readability. Once the program starts, it creates an ArrayList of type Customer and saves it by utilizing a method call to fileRead which reads the input csv file that was given. I chose to use this data structure because it has a constant time complexity when retrieving info from a known index, also the worst case complexity of it is $O(n)$. Also, utilizing an ArrayList has advantages over a normal array because it can be resized and has more methods

for more efficient data storing functionality. Once the data is stored, it prompts the user with a welcome prompt and asks the user to choose if they are a customer or a bank manager. The choices are received through a scanner and if the user inputs a number corresponding to one of the choices, it brings the user to a while loop that handles the code associated with that choice.

In the case of a user being a customer, they are prompted to log in with their Identification number. The program will verify via utilizing an enhanced for loop; if the ID number matches one in the ArrayList then it allows the user to proceed; otherwise the program ends. Once the user is verified, the program logs to a text file that they have logged in, then it prompts the customer with the main directory for a customer and asks the user to select one of the services available to them. When the customer inputs a correct selection, the program arrives to a switch statement which handles all the selection cases. I decided to use a switch statement instead of if-else statements because I found it easier to visualize as a selection menu. Case 0 handles the logout functionality. When the program arrives to this case, it logs to a text file that a user has logged out, generates an updated balance sheet of all the users, and end the program with a notification that the customer has logged out. Case 1 handles a customer inquiring about their balance. The program executes a method that will handle all of the functionality that involves this choice which includes logging to a text file upon every subsequent choice. After the method has performed its action it will print the main directory and the program will return back to main. The case will prompt the customer to select another choice pertaining to the directory. Cases 2, 3, and 4 function the same way. The functionality of the cases are inside the methods which will be called on the first line of each respective case. Case 5 works differently because the functionality is inside of the case with no method. My decision behind this is I had a difficult time passing the field Customer to methods at the time of creation. Upon implementation of the

bank manager functionality, I figured out how to solve that issue. However, I am leaving it the way it is currently to help with readability when it comes to the method that is called inside which I will elaborate on after touching on the main functionality. The function for case 5 is to send money to another individual's account. The program asks the customer to input the first name and last name of the person they are trying to send money to. The program will take the inputs of both scanners and concatenate them separated by a space in a string to compare to the names that are stored in the ArrayList. Utilizing an enhanced for-loop the program will iterate through the data that is stored. First, the program takes the first name and the last name of the current iteration and concatenate them separated by a space. If the program confirms that they are equal without respect to the case sensitivity, the program then calls a method that handles the choice making about which account the customer will send the money to.

In the case of the user being a bank manager, the program then goes to a while loop that handles the code involving the bank manager. They are not prompted to log in when the method executes; rather, they are met with a main directory that will list the current actions that they have access to. Bank managers have two options in the main directory; inquire about accounts or to log out. Choosing inquire accounts will bring the manager to another menu, asking them how they will be inquiring about the customer's accounts. They can choose to inquire by account name, account type & number, or list all the accounts. Choosing name will function the same as case 5 for a customer by verifying if the individual being searched exists in the system. If the customer is found it will print all their information. If the bank manager selects to inquire about account type & number, the program will ask to select 1-3 which account they would like to search by. Once selected, it will prompt the manager with entering in an account number that

corresponds to the category of the account type they chose to inquire by. If they select checking accounts, they must input a checking account number for the program to accept the input.

4. Testing

I used white-box testing since I developed the program starting from the classes all the way to the individual choice-making implementations. I tested my code by trying to input incorrect unexpected information in the scanners. I also tried to test the logic between switches and loops by seeing what happened if I broke out of a loop while in a switch or vice versa to ensure that the program kept running. I tested my program in different ways depending on what needed to be tested. I tested the loops that handled data retrieval of a second individual by printing the current iteration and matching it with what was given and expected. I did this to make sure that the program was viewing the given user input as intended. I selected a few users to test if the iterations were working properly. The first, middle, and last customers in the input file were the main ones to test for the purpose of verifying that retrieval from the data structure was working. I tested depositing, withdrawing, transferring, and paying by inputting negative numbers or characters to try and see where the program would fail. Most of the testing involved ensuring that the switch cases functioned as intended and making sure that if the user inputted a selection that was not in the range of choices in the directory, the program would still stay running.

I tested my user input code greatly and made sure to break the program by any means necessary, I do not think it is perfect, but my exceptions handle much better than the previous programming assignment. My testing methods can be more efficient especially in regard to testing the user input. I double or triple tested every choice prompt that the customer was given even if I knew a certain prompt was working.

Test results

I tested the deposit and withdraw methods with JUnit testing.






Test case 1 in depositTest was tested using an assertEquals that expected 16 and checked the method by depositing 6 to an initial balance of 10 which passed the test.

Test case 2 in depositTest was tested using an assertEquals that expected 9 and checked the method by depositing -1 to an initial balance of 10 which failed the test and threw a user-made NegativeNumberException.

Test case 3 in depositTest was tested using an assertNotSame which did not expect a 63 and was given a deposit of 3.5 to an initial balance of 500 which passed the test.

Test case 4 in depositTest was tested using an assertFalse which required a Boolean operation. The operation tested if depositing 73.5 to an initial balance of 40 will be greater than 1000. The Boolean operation results in a false, which is what the test case needs to pass, therefore, the test passed.

Test case 5 in depositTest was tested using an assertTrue which required a Boolean operation. The operation tested if depositing 73.5 to an initial balance of 40 will be greater than 1000. The Boolean operation results in a false, which is the opposite of what the test case needs to pass, therefore, the test failed.

depositTest			
> test1	Passed	0s	
> test2	Failed	0s	
> test3	Passed	0s	
> test4	Passed	0s	
> test5	Failed	0s	

Test case 1 in withdrawTest was tested using an assertEquals that expected 4 and checked the method by withdrawing 6 from an initial balance of 10 which passed the test.

Test case 2 in withdrawTest was tested using an assertEquals that expected 11 and checked the method by withdrawing -1 from an initial balance of 10 which failed the test and threw a user-made NegativeNumberException.






Test case 3 in withdrawTest was tested using an assertEquals that expected -1 and checked the method by withdrawing 11 from an initial balance of 10 which failed the test and threw a user-made SubtractionOverflowException.

Test case 4 in withdrawTest was tested using an assertNotSame which did not expect a 452.5 and was given a withdraw of 45.5 from an initial balance of 500 which passed the test.

Test case 5 in withdrawTest was tested using an assertFalse which required a Boolean operation. The operation tested if withdrawing 573.5 from an initial balance of 1524 will be greater than 1000. The Boolean operation results in a false, which is what the test case needs to pass, therefore, the test passed.

Test case 6 in withdrawTest was tested using an assertTrue which required a Boolean operation. The operation tested if depositing 1524.5 to an initial balance of 2379 will be greater than 1000. The Boolean operation results in a false, which is the opposite of what the test case

needs to pass,
therefore, the
test failed.

withdrawTest			
> test1	Passed	0.01s	
> test2	Failed	0s	
> test3	Failed	0s	
> test4	Passed	0s	
> test5	Passed	0s	
> test6	Failed	0s	