Wallet Simulator

Blake Bergquist

1 Introduction

Java, as an Object Oriented Programming language, allows users to represent complex, real-world items by defining data and actions that are relevant to represent the item. In Java, the data is represented using primitive type and other object variables while actions are represented using methods. In this assignment, you will be representing a basic wallet by defining the variables and methods that make up a Wallet class and a Card class.

Do not forget to put the collaboration statement at the top of your submission. See the course syllabus for more details.

2 Problem Description

Seeing all the recent technology surrounding finance (Apple Pay, Venmo, Google Wallet, etc.), you've decided that you want to design your own e-pay program. Your wallet will use a separate Card class to represent credit cards and debit cards that will then be stored in the wallet. The user will then be able to add cards, make purchases using the cards that are in the wallet, and print out the contents of their wallet via methods that you will implement.

3 Solution Description

3.1 Card Class

1. Private instance variables
   (a) A **final int cardNumber** representing the card's number. This is final because a credit or debit card's number never changes.
   (b) A **private String cardName** assigned by the user to help distinguish between cards.
   (c) A **final String cardOwner** that determines who the card belongs to.
   (d) A **final boolean isCredit** that determines whether it is a credit card or a debit card.
      i. A card has to be either a debit card or a credit card. If isCredit is true, the card is a credit card. If isCredit is false, the card is a debit card. The variable is final because a card never changes between debit and credit after it is created.
      ii. Later on, we will see a better way of modelling this situation. For now, stick to this boolean flag telling us which type of card it is.
   (e) A **double balance**. For debit cards, balance ≥ 0; for credit cards, balance ≤ 0.
2. Constructors: you must enforce the conditions for balance.

(a) public Card(int cardNumber, String cardName, String cardOwner, boolean isCredit, double balance)

3. Getters. Provide getters for the following variables using standard getter notation:

   (a) cardNumber

   (b) cardName

   (c) cardOwner

   (d) isCredit

   (e) balance

4. Public methods:

   (a) boolean updateBalance(double spent): Used for updating the balance for purchases on each card. Should enforce the conditions for balance (see validBalance under helper methods). If the new balance is a valid balance and the balance is updated, return true. If the balance is an invalid balance, the method should not update the balance and should return false.

   (b) String toString(): This should return a String that states whether it is a debit or credit card, gives the cardName, gives the cardOwner, and prints out the balance in currency format.

5. Private helper method:

   (a) boolean validBalance(double possibleNewBalance): This should return true if possibleNewBalance meets the specified criteria of debit and credit cards and false if possibleNewBalance does not. Debit cards' balance must satisfy balance ≥ 0. Credit cards' balance must satisfy balance ≤ 0

3.2 Wallet Class

1. Private instance variables

   (a) A **final String owner** that represents whose wallet the object is. The reason it is final is because the owner of a Wallet should never change.

   (b) A **String password** used to verify transactions (see validate under helper methods)

   (c) A **Card[] cards** to hold all the debit and credit cards. The array should be length 10 and will not have to be resized.

   (d) A **int numCards** to keep track of how many cards are currently in the wallet.

   (e) A **double balance**. to represent the total balance of your wallet

2. Constructors: you must utilize constructor chaining

   (a) public Wallet(String owner, String password, Card[] cards)

i. Set the instance variable cards equal to the cards array passed in here. We will be sure to only pass in arrays of size 10. If cards is null, you should make the instance variable array equal to a Card[] of size 10 with all of its elements equal to null. Be sure to update the Wallet's balance after adding all the cards (see updateBalance under helper methods).

(b) public Wallet(String owner, String password)

i. Call the other constructor, passing in owner, password, and then null for cards. This is called constructor chaining.

3. Getters. Provide getters for the following variables using standard getter notation:

   (a) balance

   (b) owner

4. Public methods:

   (a) void add(Card[] newCards): Adds additional cards to the wallet. When adding the cards, check that the cardOwner String matches the Wallet's owner before adding it. If they don't match, don't add the card. Because the wallet can only hold 10 cards, some or all of the cards may not be added to the wallet once it's full. Keep track of all cards that could not be added (whether that be because the wallet is full or the names don't match) and print out to the user all the cards that could not be added. Be sure to update (see updateBalance under helper methods) and print out the Wallet's new balance afterwards.

   (b) void buy(double price, String cardName): First, only allow the purchase to made if the owner validates (see validate under helper methods) it. Then, check if the card specified is in the wallet. If it isn't, tell the user. If it is, check to see if the purchase can be made (see updateBalance in the Card Class). If the purchase cannot be made, tell the user. If it can, update the card's balance accordingly. Be sure to update (see updateBalance in the Wallet class under helper methods) and print out the Wallet's new balance.

   (c) String toString(): The String will be each card within the wallet's toString() concatenated together

5. Private helper methods:

   (a) private void updateBalance(): Goes through and sums up all the card balances to update the Wallet's total balance

   (b) private boolean validate(): Prompts the user for the Wallet's password and returns true only if the user enters the correct, case-sensitive password. If they enter the wrong password, tell the user and return false.

3.3 WalletDriver Class

This class is what is used to test that all your methods are implemented correctly. Don't worry about what this is doing or what is in this class. As long as you implemented everything as specified, it should all work. When testing your code after writing everything else, run this class. You should still be testing small pieces of functionality as you go along (see Tips section).

4 Tips, and Tricks

- Start this early! We are getting into some of the most important topics of the course, it is imperative that you understand them completely!
- The first thing you should do for this assignment is draw out how the two classes will interact with each other. Draw boxes for each class, put in the methods required for each, and draw arrows where you'll need methods to depend on other classes and methods. This should give you a high level workflow of what each method needs to do.
- Make sure you understand the constraints of certain methods/values. Balance has a different constraint depending on if the card is a credit or debit card; it needs to be correctly upheld at all times!
- Start small and test individual pieces of your code. You won't be able to run the driver until you have most of your classes set up, but that doesn't mean you can't write your own driver to test certain functionality early!

5 Checkstyle

Checkstyle counts for this homework. You may ignore the: WalletDriver.java:4:5: Method length is 244 lines (max allowed is 150). error. You may be deducted up to 20 points for having Checkstyle errors in your assignment. Each error found by Checkstyle is worth one point. This cap will be raised next assignment. Again, the full style guide for this course that you must adhere to can be found by clicking **here**.

Come to us in office hours or post on Piazza if you have specific questions about what Checkstyle is looking for and how to fix Checkstyle errors.

First, make sure you download the `checkstyle-6.0-all.jar` and `cs1331-checkstyle.xml` from the T-Square assignment page. Then, make sure you put *both* of those files in the same directory (folder) as the `.java` files you want to run Checkstyle on. Finally, to run Checkstyle, type the first line into your terminal while in the directory of your Java files and press enter.

```
$ java -jar checkstyle-6.0-all.jar -c cs1331-checkstyle.xml *.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. You can easily count Checkstyle errors by piping the output of Checkstyle through `wc -l` and subtracting 2 for the two non-error lines printed above (which is how we will deduct points). For example:

```
$ java -jar checkstyle-6.0-all.jar -c cs1331-checkstyle.xml *.java | wc -l
      2
```

Alternatively, if you are on Windows, you can use the following instead:

```
C:\> java -jar checkstyle-6.0-all.jar -c cs1331-checkstyle.xml *.java | findstr /v "Starting
    audit..." | findstr /v "Audit done" | find /c /v "hashcode()"
0
```

6 Turn-in Procedure

Submit your `Wallet.java` and `Card.java` files on T-Square as an attachment (you may also submit `WalletDriver.java` if you want to make our lives easier, but are not required to). Do not submit any compiled bytecode (`.class` files), the Checkstyle jar file, or the `cs1331-checkstyle.xml` file. When you're ready, double-check that you have submitted and not just saved a draft.

7 Verify the Success of Your Submission to T-Square

Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.

2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.

3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.

4. Recompile and test those exact files.

5. This helps guard against a few things.

   (a) It helps insure that you turn in the correct files.

   (b) It helps you realize if you omit a file or files.[1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)

   (c) Helps find last minute causes of files not compiling and/or running.

[1] Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is 8PM Thursday. Do not wait until the last minute!