Final Programming Project

Due: Friday, December 8, by 11:59 PM. Upload your source code through Blackboard. ***Add your name and Stony Brook ID number at the top of each file.***

CSEBank has decided to offer a new collection of credit cards to its customers: a basic/traditional credit card, a rewards card, and a prepaid/stored-balance card. For this assignment, you will develop a program that processes a data file containing a list of customer transactions and prints a monthly statement for each card.

Follow the instructions below ***carefully;*** they are extensive, but they provide a highly-detailed description of how to complete each step of the development process.

Part 1: Laying The Foundations

1. To begin with, we need to develop classes to represent the different card types. These classes share various properties, so inheritance is extremely useful here. For example, a `RewardsCard` is a specialized type of `CreditCard`. A `PrepaidCard` is similar to a `CreditCard`, but not identical, so it should be a sibling that shares a common `BankCard` parent class.

   We have provided a fully-implemented `Transaction` helper class that represents a single card transaction (e.g., a credit or debit). While you should inspect the source code for this class yourself to see exactly how it works, a quick summary of its public interface methods is given below:

   • A single constructor: `Transaction(String type, String merchant, double amount)`

   • Four (no-argument) accessor methods: `type()`, `amount()`, `merchant()`, and `notes()`

   • Two mutator methods: `setNotes()` and `addNotes()`, each of which takes one `String` argument

   • A `toString()` method

   We have also given you source code for three test driver programs to make sure that your `BankCard` subclasses work correctly before you proceed to Part 2 (the main program). If you plan to use these drivers to double-check your code (not strictly mandatory, but ***HIGHLY*** recommended!), be sure that each of your classes uses the ***exact*** method headers described below; otherwise, the test drivers won't work (at least, not without some significant modification).

   Every card issued by CSEBank has a 16-digit number whose first two digits indicate its type:

   a. Credit cards have numbers that begin with 84 or 85.

   b. Rewards cards have numbers that begin with 86 or 87.

   c. Prepaid cards have numbers that begin with 88 or 89.

2. Start by defining an `abstract` `BankCard` class with the following elements:

   a. Instance variables:

      i. a `private` `String` variable that stores the cardholder's name

      ii. a `protected` `long` variable that holds the card number (a 16-digit integer)

      iii. a `protected` `double` variable that stores the card's current balance

      iv. a `protected` `ArrayList` that holds `Transaction` objects

   b. Constructors and instance methods (***DO NOT*** modify the headers/signatures specified below):

      i. `public BankCard(String cardholderName, long cardNumber)`. This constructor sets the initial card balance to 0.

      ii. `public double balance()`. This method returns the current balance of the card.

      iii. `public long number()`. This method returns the card number.

      iv. `public String cardHolder()`. This method returns the cardholder's name.

      v. `public String toString()`. This method returns a `String` containing (at minimum) the card number and the current balance, with appropriate labels, as in

         Card # 1234567890123456 Balance: $1234.56

      vi. `public abstract boolean addTransaction(Transaction t)`. When implemented, this method attempts to performs the specified `Transaction` on a card. Its return value indicates success or failure.

      vii. `public abstract void printStatement()`. When implemented, this method prints the current card's basic information (cardholder name, card number, balance, etc.), along with a list of transactions that have been completed successfully.

3. The `CreditCard` class extends `BankCard`. It contains the following elements:

   a. Instance variables:

      i. a `private` integer variable that stores the card's expiration date: a 3- or 4-digit integer in *myy* or *mmyy* form. Note that single-digit months ***CANNOT*** be preceded with a leading 0; if they are, Java will assume that the integer is written in octal (base 8) instead of decimal (base 10) as intended.

      ii. a `protected double` variable that stores the card's credit limit (in dollars)

   b. Constructors and instance methods:

      i. `public CreditCard(String cardHolder, long cardNumber, int expiration, double limit)`. For now, you may assume that the card number is always valid.

ii. `public CreditCard(String cardHolder, long cardNumber, int expiration)`. This constructor sets the credit limit to $500.00. For now, you may assume that the card number is always valid.

iii. public double `limit()`. This method returns the credit limit for this `CreditCard`.

iv. public double `availableCredit()`. This method returns the difference between the credit limit and the current balance (i.e., *limit* - *balance*).

v. `public int expiration()`. This method returns the expiration date for this `CreditCard`.

vi. `public String toString()`. This method overrides `BankCard`'s version, returning a `String` with the card number, its expiration date, and its current balance.

vii. `public boolean addTransaction(Transaction t)`. This method implements the `abstract BankCard` method. It returns a `boolean` value according to the rules below:

1. If the `Transaction` type is "debit" and the transaction amount is less than or equal to this card's available credit, the transaction is accepted; its amount is ***added to*** the card balance, the `Transaction` is added to the card's `ArrayList` of transactions, and the method returns `true`.
2. If the `Transaction` type is "debit" and its amount is greater than this card's available credit, the transaction is declined; the `ArrayList` does not change, and the method returns `false`.
3. If the `Transaction` type is "credit", the transaction amount (which will be a negative number) is added to the card balance (lowering it), the `Transaction` is added to the card's `ArrayList` of transactions, and the method returns `true`.
4. If the `Transaction` has any other type (like "redemption"), the `ArrayList` does not change, and the method returns `false`.

viii. `public void printStatement()`. This method implements the `abstract` method from `BankCard`. It prints out the card's basic information (cardholder name, card number, expiration date, and ending balance), along with a neatly-formatted list of card transactions.

4. The `RewardsCard` class extends `CreditCard`. It contains the following elements:

a. Instance variables:

i. a `protected` integer variable that stores the current number of available reward points

b. Constructors and instance methods:

i. `public RewardsCard(String holder, long number, int expiration, double limit)`. This constructor initializes the reward points to 0. For now, you may assume that the card number is always valid.

ii. `public RewardsCard(String holder, long number, int expiration)`. This constructor sets the credit limit to $500.00 and initializes the reward points to 0. For now, you may assume that the card number is always valid.

iii. `public int rewardPoints()`. This method returns the current number of reward points.

iv. `public boolean redeemPoints(int points)`. This method operates as follows:

1. If the parameter is less than or equal to the current number of reward points:

    a. Reduce the card balance by an amount equal to the number of points redeemed divided by 100.00 (for example, 1255 reward points can be used to lower the card balance by $12.55)[1]

    b. Subtract the number of points redeemed from the current number of reward points

    c. Create a new `Transaction` and add it to this card's list of transactions. The new `Transaction` has the type "redemption". Its amount is equal to the value of the balance reduction you just calculated (the `Transaction` class will automatically make this value negative to reflect that it is a form of credit). Use "CSEBank" as the merchant name. Leave the notes field empty.

    d. Return `true` to indicate success.

2. Otherwise, do not change the current balance, number of accumulated reward points, or the list of transactions. Return `false` to indicate failure.

v. `public String toString()`. This method returns a `String` containing the same data as the `CreditCard` version, plus the current number of available reward points.

vi. `public void addTransaction(Transaction t)`. This method overrides the version inherited from `CreditCard`. Its behavior matches the inherited version, except successful "debit" transactions also generate reward points equal to 100 times the transaction amount (cast to an integer). For example, a successful debit of $12.34 earns 1234 reward points.

vii. `public void printStatement()`. Like its superclass equivalent, this method prints out a neatly-formatted list of card transactions, along with the rest of the card's information (don't forget to include the reward points as well!).

5. The `PrepaidCard` class extends `BankCard`. It defines the instance methods below (it does not add any new instance variables). Note that several of these operations are somewhat different from their counterparts in `CreditCard` and `RewardsCard`!

    a. `public PrepaidCard(String cardHolder, long cardNumber, double balance)`. For now, you may assume that the card number is always valid.

    b. `public PrepaidCard(String cardHolder, long cardNumber)`. This constructor sets the starting balance to 0. For now, you may assume that the card number is always valid.

    c. `public boolean addTransaction(Transaction t)`. This method works as follows:

    i. If the `Transaction` type is "debit" and the transaction amount is less than or equal to this card's balance, the transaction is accepted; ***subtract*** its amount from the card balance, add the `Transaction` to the `ArrayList` of transactions, and return `true`.

[1] Note that this may bring the card balance below 0. This is acceptable for the basic assignment; one of the extra-credit options asks you to fix this.

   ii. If the `Transaction` type is "debit" and its amount is greater than the card balance, the transaction has been declined. Do not change the card balance or the `ArrayList`; just return `false`.

   iii. If the `Transaction` type is "credit", ***subtract*** its amount from the card balance (credits have negative amounts, so this effectively adds money back to the `PrepaidCard`), add the `Transaction` to the `ArrayList` of transactions, and return `true`.

   iv. If the `Transaction` has any other type (like "redemption"), do not change the card balance or the `ArrayList`; just return `false`.

d. `public boolean addFunds(double amount)`. This method works as follows:

   i. If the parameter is positive, it is added to the card's balance, and a new `Transaction` is added to the `PrepaidCard`'s `ArrayList` with a type of "top-up", a merchant name of "User payment", an amount equal to the argument, and no notes. Return `true` to indicate success.

   ii. Otherwise, do not modify the balance or the transaction list. Return `false` to indicate failure.

e. `public String toString()`. This method returns a `String` that contains, at minimum, the card number and its current balance.

f. `public void printStatement()`. This method implements the `abstract` method from `BankCard`. It prints out a neatly-formatted list of card transactions, along with the rest of the card's information (cardholder name, card number, and ending balance).

6. Define a new `CardList` class. This class contains the following elements:

   a. Instance variables:

      i. a `private ArrayList` that can hold `BankCard` objects (including any of its subclasses).

   b. Constructors and instance methods:

      i. `public CardList()`. This constructor assigns a new (empty) `ArrayList<BankCard>` object to the instance variable above.

      ii. `public void add(BankCard b)`. This method adds its parameter to the internal `ArrayList` (you may add at any fixed position you want; we suggest the end for simplicity).

      iii. `public void add(int index, BankCard b)`. Index values are 0-based, just like traditional arrays and `ArrayLists`. If the specified index is between 0 and the internal `ArrayList`'s current size (inclusive), this method inserts the `BankCard` at that index. Otherwise, the `BankCard` is added at the end of the `ArrayList`. **This method is *OPTIONAL;* you are *NOT* required to implement it for this program** (although it may be nice to have)!

      iv. `public int size()`. This method returns the current size of the internal `ArrayList` (meaning the number of elements it contains).

      v. `public BankCard get(int index)`. Index values are 0-based, just like traditional arrays and `ArrayLists`. If the specified index is valid, this method returns a reference to the

BankCard at that index within the internal ArrayList (it does not alter the ArrayList's contents in any way). Otherwise (meaning that the index value is negative or greater than or equal to the CardList's current size), this method returns null.

vi. public long indexOf(long cardNumber). This method uses linear search (or a different search technique, if you wish, as long as it works correctly) to return the (0-based) index of the first (and ideally only) BankCard in the ArrayList whose card number matches the parameter value. If the ArrayList does not contain a BankCard object with a matching card number, this method returns -1.

Part 2: Implementing The Main Program

Now that our program foundations are (finally!) complete, it's time to develop the main program. Our program will perform the following tasks (full details are provided below, including the specific format for each type of data file):

c. Prompt the user for and read in the name of a (plain text) data file containing information describing a number of BankCards. Each line of the data file describes one BankCard.

d. Read and parse the contents of this file to create and populate (fill in) a new CardList.

e. Prompt the user for and read in the name of a (plain text) data file containing a list of transactions. Each line of the file describes a separate transaction.

f. Process this second file, one line at a time. For each transaction, locate the matching card in your CardList and apply the transaction to it.

g. After the transaction file has been completely processed, display a final monthly statement for each card in your CardList.

We have provided a program skeleton (TransactionProcessor.java) for you to complete this part of the project. It includes a method named getCardType(), which returns a String identifying the type of card that a given card number represents. Complete the TransactionProcessor class as follows:

1. Add a method with the header:

   ```
   public static BankCard convertToCard(String data)
   ```

   This method uses a Scanner to break up the input String into a long (the card number), a String (the cardholder name), and possibly a double (the card limit) followed by an int (the expiration date). It returns a subclass of BankCard built using that data:

   a. Start by creating a Scanner to read from the parameter. Recall that you can initialize a Scanner with a String as its argument instead of the more commonly-used System.in. If you do this, the Scanner will only collect input from that string, rather than from standard input (the keyboard). Use the Scanner method nextLong() to retrieve the card number.

   b. Use the provided getCardType() method to get a String corresponding to the type of card this data belongs to. If the card type is null, skip the next step and move to the next line of the file.

   c. Using an if-else chain, determine which of the three BankCard subclasses you are dealing with. Then read the remaining fields from the parameter (see the end of this document for details on the data file format) and create and return a new card object of the appropriate type.

2. Add a method with the header:

   ```
   public static CardList loadCardData(String fName)
   ```

   This method takes the name of a card data file as its argument. It opens and processes that file, one line at a time, before returning a `CardList` of `BankCard`s based on the data file contents (you may add the cards to the `CardList` in any order you want, not necessarily the order they are listed in the card data file). Use the `convertToCard()` method you defined in the previous step to process each individual line of the data file. **If you encounter into an `IOException`, your method should return `null` to indicate failure.**

3. Add a method with the header:

   ```
   public static void processTransactions(String filename, CardList c)
   ```

   This method opens the card transaction file named *filename* for reading. For each line in the file:

   a. Use the `String` method `split()` to divide the line into an array of smaller substrings (use a single space as the separator/delimiter).

   b. Use the `Long.parseLong()` utility method to convert the first substring (the card number) from a `String` into a `long` value.

   c. Find the index of the card with that card number in your `CardList`.

   d. The second substring indicates what type of transaction we are working with (you may assume that all transactions are performed on the correct type of card):

      i. If the transaction type is "redeem":

         1. Convert the third substring to an integer value (the number of points to redeem) with `Integer.parseInt()`.

         2. Retrieve the card at the specified index and cast it back to a `RewardsCard`.

         3. Call the card's `redeemPoints()` method with the number of points.

      ii. If the transaction type is "top-up":

         1. Convert the third substring to a `double` value (the amount of money to add) with `Double.parseDouble()`.

         2. Retrieve the card at the specified index and cast it to a `PrepaidCard`.

         3. Call the card's `addFunds()` method with the top-up value.

      iii. Otherwise:

         1. Create a new `Transaction` from the second, fourth, and third substrings (the third substring should be converted to a `double` value using `Double.parseDouble()`).

Leave this `Transaction`'s notes field empty.

2. Retrieve the card at the specified index (as a `BankCard` object).

3. Call that card's `addTransaction()` method with the `Transaction` object you just created.

e. If you encounter an `IOException`, stop processing the file and return immediately from the method (do not try to reverse any transactions that you have already processed successfully).

4. Add a `main()` method to your `TransactionProcessor` class. `main()` should use your helper methods from the previous steps to perform the following sequence of tasks:

   a. Read in the name of the card data file from the user.

   b. Use `loadCardData()` to create a `CardList` from the contents of the card data file.

   c. If `loadCardData()` successfully creates a list of cards:

      i. Read in the name of a transaction data file from the user.

      ii. Use `processTransactions()` to read the contents of the transaction file and apply them to the list of cards.

      iii. Once you have finished processing the transaction file, print out a summary statement for each card in the `CardList` (using its `printStatement()` method).

Data File Formats

The card data file is a plain text file with no blank lines. Each line of the file has two required values (the card number and cardholder name, in that order), possibly followed by one or two optional values. There is a single space between each field:

card_number cardholder (for a `PrepaidCard` with the default starting balance)

or

card_number cardholder expiration (for a `CreditCard` or `RewardsCard` with the default credit limit)

or

card_number cardholder balance (for a `PrepaidCard` with a specific starting balance)

or

card_number cardholder expiration limit (for a `CreditCard` or `RewardsCard` with a specified credit limit)

Every field has a specific type:

- *card_number* is a `long`.
- *cardholder* is a `String` where all spaces have been replaced by underscore ('_') characters (so, for example, "John Smith" will be represented as "John_Smith").
- *expiration* is an `int`.
- *balance* and *limit* are both of type `double`.

For example:

```
8434567812345678 John_Smith 318 12345.67
```

represents a `CreditCard` whose expiration date is March 2018 and whose credit limit is $12345.67.

Likewise,

```
8902345682910224 Mary_Doe
```

represents a `PrepaidCard` issued to "Mary Doe" with an initial balance of $0.

The transaction data file is a plain text file with no blank lines. Each line of the file begins with two required fields (the card number and transaction type, in that order), followed by one or two additional fields (depending on the transaction type). Each field is separated by a single space.

The card number is a `long` value.

There are four transaction types (each of which is a `String`): *redeem*, *top-up*, *debit*, and *credit*

redeem transactions have a third field (an `int`) representing the number of points being redeemed.

top-up transactions include a third field (a `double`) representing the amount of money to add.

debit and *credit* transactions include two additional fields representing the transaction amount (a `double`) and the merchant name (a `String` where all spaces have been replaced by underscore ('_') characters), respectively.

Sample transaction records:

```
8634567812345678 redeem 2048

8853946610327745 top-up 500.00

8734621893246154 debit 397.45 Amazon

8419567820318835 credit -98.72 Gas_N_Go
```