CS 213 Object Oriented Programming
Assignment 4

Total Points:	40
Assigned:	Thursday, March 16, 2023
Due:	Friday, March 24, 2023, by 11:59pm
Submission format:	Electronically, via Canvas

Resources

Some resources have been provided to assist in the assignment. You may find them below:

- this assignment in a .pdf file here
- Assignment 3 solution here
- Base/started code for Assignment 4 (this one) here

Instructions

- Work individually on the assignment. Remember that you are allowed to brainstorm with your peers, but each person must write their code individually
- **Remember** that the purpose of the assignment is to give you a better understanding of the concepts we are learning. So make sure you start the assignment early and focus on **fully understanding** what you are doing, asking questions and seeking clarification as you go along, from the course instructors and faculty interns. Do not be satisfied with simply having working code. To meet the learning objectives of the assignment, you must understand what you have done and why!
- The material in Chapters 5, 6, and 7 of the textbook covering objects and classes as well as arrays, will be useful for this assignment

Problem 1 [5 points]: Reflecting on Assignment 3

Download the solution to Assignment 3 (the enhanced Person class), and study it carefully, comparing it with your own solution.

- In places where your solution differs from the provided solution, do some research and/or discuss with classmates or the course instructors to understand whether both approaches are valid or if there's a shortcoming in your approach
- If your solution had shortcomings, try to pinpoint what you misunderstood – for example, was it a misunderstanding of a particular programming concept, or a misunderstanding of the assignment question? Has that misunderstanding now been clarified?

Write and submit in a word/PDF document a paragraph or two reflecting on what you have learned from this exercise.

Problem 2 [15 points]: Creating a Bus class

Define a class called **Bus** that represents a bus that holds a given number of people.

A. The bus capacity should be an instance variable initialized to the value of a parameter given to the constructor of the class. Other instance variables include the bus license plate number, the bus make and model, and the bus colour. The Bus should have an instance variable to represent the driver (a Person object) and it should use an array of Person objects to represent the passengers in the bus. It should keep track of the number of passengers currently on the bus.

Write the following methods for your Bus class. Also, create a test program (e.g. in a file called TestBus), which fully tests the class. Remember to test incrementally as you write your code.

B. A **constructor** that takes parameters to represent the bus capacity, license plate number, make, model, and colour. It sets the instance variable for the driver to null and creates an empty array for the passengers.
C. **Accessor methods** for the bus capacity, license plate number, make, model and colour.
D. A method **setDriver()** which takes as input a reference to a Person object. If the person is above the driving age (I.e. the canDrive() method of the Person object returns true), the bus's driver is set, replacing any driver that was there previously. The method then returns true. Otherwise, it returns false.
E. A method **hasDriver()** which returns a boolean indicating whether or not the bus currently has a driver.
F. A method **addPassenger()** that takes as input a reference to a Person object. The method tries to put the passenger in an empty slot in the array of passengers and returns whether or not it was successful in doing so. Do not forget to increment the number of passengers if the passenger was successfully added.
G. A method **removePassenger()** that takes as input a reference to a Person object. The method goes through the array of passengers looking for a passenger that is "equal" to the given passenger. If such a passenger is found, the passenger is removed (i.e. that slot of the array is set to null) and the method returns true. Do not forget to decrement the number of passengers. If such a passenger is not found, the method returns false.
H. A method **getNumPassengers()** that returns the current number of passengers on the bus
I. A method **getNumOccupants()** that returns the current number of occupants (including the driver)
J. A method **hasPassengers()** which returns whether or not the bus has at least one passenger
K. A method **isEmpty()** which returns whether or not the bus is empty (has no driver or passengers)
L. A method **isFull()** which returns whether or not the bus is full (all spots for the driver and passenger are taken)

M. A method **listRiders()** which displays the name, gender and age of the driver and each of the passengers in the bus
N. A method **getEmptySeats()** which returns an array of the empty passenger seat numbers in the bus. The seat number is simply represented by the index in the array of passengers. So essentially, this method returns an array containing the indices of the empty slots in the array of passengers. For example, consider a bus with a capacity of 10 passengers that has passengers are indices 0, 1, 2, 5, 6, and 9. The getEmptySeats() method will return an array containing the integers 3, 4, 7 and 8 because these are the indices in the passengers array that do not have passenger objects. Thus, they represent the "empty seats".

Problem 3 [20 points]: Implementing a BankAccount class

In this assignment, you will implement a class that simulates a bank account. The class has been started for you (see BankAccount.java). Its member variables, defined constants, and one constructor have been defined for you. You will need to implement the remaining methods of the class.

Note that the BankAccount class defines an *enumeration*, AccountType that defines constants for the two types of accounts that may be created (CURRENT and SAVINGS). We've mentioned enumerations in class and you were asked to read about them in the textbook a while ago. If you have not yet done so, now would be a good time to do so. Let us know if you have any questions.

The BankAccount class defines several member variables including the type of account (AccountType *acctType*), the account's identifier or ID (String *acctID*), the current balance of the account (double *balance*), the number of withdrawals that have been made in the current month (int *numWithdrawals*), and whether or not the account is currently "in the red" (boolean *inTheRed*). An account that is "in the red" is one whose current balance is less than the required minimum balance in the account. The required minimum balance is stored in a member variable (double *minBalance*). The member variable representing the minimum balance is initialized to the value of one of two defined constants, CURRENT_ACCT_MIN_BALANCE or SAVINGS_ACCT_MIN_BALANCE, depending on whether the account is a current account or a savings account.

A bank account may also earn interest. The annual interest rate (which may be 0 if the account does not earn interest) is stored in a member variable (double *interestRate*). The earned interest is computed monthly. An account also has a monthly maintenance fee (double *maintenanceFee*) which is a fee that is deducted every month from the account balance and may be 0. Lastly, some accounts have a limit on the number of withdrawals that can be made in a month, and this limit is also stored in a member variable (int *withdrawalLimit*). Note that if there is no withdrawal limit, we set the *withdrawalLimit* member variable to -1.

In our program, savings accounts attract interest, and the rate is specified by a defined constant SAVINGS_ACCT_INTEREST_RATE. Current accounts do not attract interest. Current accounts have a monthly maintenance fee, specified by the defined constant CURRENT_ACCT_MAINTENANCE_FEE. Savings accounts do not have a maintenance fee. Lastly, savings accounts have a withdrawal limit of 2 withdrawals a month, specified by the defined constant SAVINGS_WITHDRAWAL_LIMIT. Current accounts do not have a withdrawal limit.

Remember that a *constructor* is a method that is used to initialize the member variables of a class. Note that a class may have more than one constructor, provided each one has a different list of input parameters. Note that we have defined one constructor for you in the BankAccount class:

> **Method**: constructor
> **Input parameters**: AccountType *type*, String *id*
> **Description**: Initializes the bank account. Sets the current balance to zero and the number of withdrawals for the current month to 0. If the AccountType is CURRENT, it sets the minimum balance to the defined constant CURRENT_ACCT_MIN_BALANCE, the interest rate to 0, and the maintenance fee to CURRENT_ACCT_MAINTENANCE_FEE. Otherwise, the account must be a savings account, and so it sets the minimum balance to the defined constant SAVINGS_ACCT_MIN_BALANCE, the interest rate to SAVINGS_ACCT_INTEREST_RATE, and the maintenance fee to 0. The initial value of the *inTheRed* member variable is set based on the current balance and the specified minimum balance.

You will define the remaining methods of the bank account class. As you define each method, you should test it by adding relevant method calls to the TestBankAccount class that has been provided. Turn in your modified TestBankAccount.java source file with the assignment – we would like to see how you tested your BankAccount class.

First, define another constructor for the bank account class that takes in an opening account balance, in addition to an account type and id:

A. **Method**: constructor
   Input parameters: AccountType *type*, String *id*, double *openingBalance*
   Description: Initializes the bank account. The behavior of this constructor is mostly the same as the previous constructor that has been defined, except that the current balance is initialized to the specified *openingBalance*, rather than to 0.

Next, define and implement the following methods for the BankAccount class:

B. **Method**: getBalance
   Return type: double
   Input parameters: none
   Description: returns the current account balance.

C. **Method**: getAccountType
Return type: AccountType
Input parameters: none
Description: returns the account type.

D. **Method**: getAccountID
Return type: String
Input parameters: none
Description: returns the account id.

E. **Method**: getMinBalance
Return type: double
Input parameters: none
Description: return the minimum balance for the account

F. **Method**: withdraw
Return type: boolean
Input parameters: double *amount*
Description: This method tries to deduct the specified *amount* from the account balance and increments the number of withdrawals. If the transaction is successful, it returns true. Otherwise, it returns false. The transaction will be unsuccessful if the number of allowed withdrawals has been exceeded, the bank account is "in the red" or the bank account does not have a sufficient balance (i.e. removing the amount would leave less than the required minimum balance in the account). If the transaction is not successful, the method should print out an explanation before it returns. E.g.:

Sorry, could not perform withdrawal: Insufficient balance.

Remember that when there is no withdrawal limit, the *withdrawalLimit* member variable is set to -1, so your code must check for this case and behave appropriately.

G. **Method**: deposit
Return type: void
Input parameters: double *amount*
Description: This method adds the specified *amount* to the account balance.

H. **Method**: performMonthlyMaintenance
Return type: void
Input parameters: none
Description: This method performs monthly maintenance on the account. It computes any interest earned on the account (given the interest rate on the account) and adds the earned interest to the current balance. Note that the *interestRate* member variable specifies the annual interest rate on the account. As such, it will need to be divided by the number of months in the year in order to compute the interest for one month. This method also deducts the account's monthly maintenance fee (specified by the *maintenanceFee* member variable) from

the current account balance. After subtracting the maintenance fee, the method determines whether or not the account is in the red (i.e. if the account balance is less than the minimum balance). The method resets the number of withdrawals for the current month to 0. Lastly, the method prints out a summary of the monthly maintenance process, in the following format. (Note that items in triangular braces <> should be replaced by the appropriate values

Earned interest: <earned interest>
Maintenance fee: <maintenance fee>
Updated balance: <new balance>

If the account is in the red, the following warning should also be printed out:
WARNING: This account is in the red!

The last method you will implement is a little different from the previous ones, in that it takes in another BankAccount object as an input parameter. Remember that this is not unusual – we have seen a similar situation when we implement the equals() method for a class. Remember that if you want to refer to the member variables or methods of the BankAccount object passed in as an input parameter, simply precede them with the object name, e.g. *otherAccount*.deposit(). Note that if you omit the object name, or use the keyword *this*, e.g. deposit() or *this*.deposit(), you are referring the member method of the current (this) BankAccount object.

I. **Method**: transfer
Return type: boolean
Input parameters: boolean *transferTo*, BankAccount *otherAccount*, double *amount*
Description: This method performs a transfer between this bank account and another bank account (specified by the input parameter *otherAccount*) and returns whether or not the transaction was successful. The amount being transferred is specified by the input parameter *amount*. If the input parameter *transferTo* is true, the amount is withdrawn from this bank account and deposited into the other bank account. On the other hand, if the input parameter *transferTo* is false, the amount is withdrawn from the other bank account and deposited into this bank account. Note that you should implement this method by making calls to the previously defined withdraw() and deposit() methods. If the withdrawal from the account the money is coming from is not successful (e.g. due to insufficient balance or exceeding the withdrawal limit), then no transfer is performed and the method returns false. Otherwise, the transfer is successful and the method returns true.

Rubric

Each programme on this assignment will be graded according to a scaled version of the following rubric which is being used for most programming tasks this semester:

Points	Description
9 – 10	Exceptionally efficient, well designed, well structured, well formatted and well documented program. Elegant algorithms and creative solution approaches. Thorough and well documented testing.
8	Correct, well-structured and formatted program with good documentation and well-chosen test cases.
7	Working programme with minor shortcomings in correctness, structure, formatting, documentation and/or testing
6	Working programme with notable shortcomings in correctness, structure, formatting, documentation and/or testing
5	Good attempt, but major shortcomings in correctness, structure, formatting, documentation and/or testing
2 – 4	Submitted programme compiles and makes logical but incomplete attempts to address some aspect(s) of the problem
1	Submitted programme does not compile but makes logical attempts to address some aspect of the problem
0	Did not submit the assignment