Lab10

Exercise 1

In this exercise, we create following three classes

1. `BankAccount` as described below.
2. `SavingAccount` extends (publicly inherits) `BankAccount` and overrides necessary functions.
3. `CurrentAccount` extends (publicly inherits) `BankAccount` and overrides necessary functions.

Here is a typical client code using these classes

```
#include <iostream>
#include "bankaccount.h"
#include "savingaccount.h"
#include "currentaccount.h"

using namespace std;

int main() {

    SavingAccount a1("Sunil", 2000);
    CurrentAccount a2("Pankaj", 1000);

    try {
        a1.deposit(5000);
        a1.withdraw(3000);
        cout << "Account-1 Balance: " << a1.getBalance() << endl;
        a1.withdraw(2000);
        //a1.deposit(1500);
        a2.setOD_Limit(10000);
        cout << "Account-2 Balance: " << a2.getBalance() << endl;
        a2.withdraw(5000);
        a2.deposit(3000);
        a2.deposit(3000);
        a2.deposit(3000);
        a2.deposit(3000);
        a2.withdraw(10000);
        cout << "Account-2 Balance: " << a2.getBalance() << endl;
        a2.eom();
        cout << "Account-2 Balance: " << a2.getBalance() << endl;
        //a1.withdraw(2000);
    }
    catch(InSufficientBalance& e) {
        cout << "Error: " << e.what() << endl;
    }
    catch(TransactionLimitExceeds& e) {
        cout << "Error: " << e.what() << endl;
    }
    catch(...) {
        cout << "Unkown Error Occurred! " << endl;
    }
    return 0;
```

```
}
```

Note that, you also require creating two exception classes `InSufficientBalance` and `TransactionLimitExceeds`. These class extend (publicly inherit) `std::logic_error` class.

Functional Description of `BankAccount` class.

You can take and modify `BankAccount` class that we have already discussed in lectures. The proposed `BankAccount` class provides following operations.

- Fields
    1. account number //autogenerated
    2. customer name
    3. balance
- Constructors
    1. `BankAccount()`. account no autogenerated. customer name empty, init balance = 0.
    2. `BankAccount(cust_name)`. initial balance set to zero.
    3. `BankAccount(cust_name, ini_balance)`
- Field accessor, mutator, and other methods
    4. getter method for account number
    5. getter/setter methods for Customer Name
    6. getter method for balance
    7. `deposit(amount)`. Balance gets increased by parameter amount. Derived classes may override this method therefore let it be a virtual function.
    8. `withdraw(amount)`. Balance gets decreased by parameter amount. If amount to be withdrawn is larger than balance, method throws exception `InSufficientBalance`. Exception object should carry account number and balance information of the account. Derived classes may override this method therefore let it be a virtual function.
    9. `withdraw_od(amount)`. This basically withdrawal overdraft. that is, it does not check for availability of balance. It allows withdrawing even if there is no balance. That means it never throws `InSufficientBalance` exception. This is to be used as helper function and hence private. However it shall be used in derived classes as well therefore let this function be `protected`.
    10. `toString()`. returns string object that contains formatted all details of the account.
    11. `eod()`. Called on every day end. Let this function to be virtual with empty implementation. Derived classes are may be overriding this function.
    12. `eom()`. Called on every month end. Let this function to be virtual with empty implementation. Derived classes are may be overriding this function.
    13. `getAccountType()`. An account can be of different types. Its type is set at appropriate time. The function returns Account Type as `short`. Let it be 1 for Saving account and 2 for Current Accounts. Let there be no implementation here and be it to be virtual.

Functional Description of `SavingAccount` class.

Note: `SavingAccount` **class extends** `BankAccount` .

- Additional fields
    1. Annual Interest Rate. This rate by which account is paid interest at the end of month.
    2. Daily Transaction Count. A limited number of deposit and withdrawals can only be performed in a saving account. So a daily counter is maintained. With every withdrawal or deposit, the counter is incremented by 1. An End of Day (EOD) function is run at the

end of the day that resets counter to 0.
Let default Interest Rate and Daily Daily Transaction Limit be specified as class level constants. You can define their values 4.5, and 3 respectively.

- Constructors
    1. `SavingAccount()`. constructs with autogenerated account number. empty customer name, initial balance = 0, and default interest rate.
    2. `SavingAccount(cust_name)`. constructs with autogenerated account number. initial balance = 0, and default interest rate.
    3. `SavingAccount(cust_name, ini_balance)`. constructs with autogenerated account number. and default interest rate.
- Methods
    7. getter and setter methods for interest rate.
    8. `deposit(amount)`. Balance gets increased by parameter amount. Increments transaction count. deposit is not permitted when daily transaction number exceeds and throws exception `TransactionLimitExceeds(` instead.
    9. `withdraw(amount)`. Balance gets decreased by parameter amount. Increments transaction count. This also throws exceptions `InSufficientBalance` and `TransactionLimitExceeds(` on respective event.
    10. `payInterest()`. Account is paid for a **minimum** in the month. You may have to keep track of minimum balance for the month. The function is called at the end of month.
    11. `toString()`. Override appropriately, so that can be used for printing all details of the current bank account.
    12. `eod()`. Overrides. Calls on every end of the day, resets transaction counter to zero or so.
    13. `eom()`. Overrides. Called on every month end. Calls `payInterest()`, and initializes counters that are applicable.
    14. `getAccountType()`. Overrides appropriately.

Functional Description of `CurrentAccount` class.

Note: `CurrentAccount` **class extends** `BankAccount` .

- Additional fields
    1. Monthly transaction count. Current accounts have certain number of transactions free in a month and chargeable after that. Therefore we require keeping track of number of transaction in the month. An EOM function is run on monthly basis and that charges amount from the account for additional transaction if any.
    Let number of Free Transactions and charges per additional transaction be same for all accounts be defined as class level variables.
    2. Current accounts also have concept of over draft. A current is allowed some overdraft limit. That is the amount that can be used to permit extra withdrawal from the account beyond the balance. That means account can have negative balance up-to their overdraft limit.
- Constructors
    1. `CurrentAccount()`. constructs with autogenerated account number. empty customer name, initial balance = 0, and default interest rate.
    2. `CurrentAccount(cust_name)`. constructs with autogenerated account number. initial balance = 0, and default interest rate.
    3. `CurrentAccount(cust_name, ini_balance)`. constructs with autogenerated account number. and default interest rate.

- Methods
    7. `deposit(amount)`. Balance gets increased by parameter amount. Increments transaction count.
    8. `withdraw(amount)`. Balance gets decreased by parameter amount. Increments transaction count. Withdrawal of this class permits over withdrawal up-to overdraft limit of the account.
    9. `deductCharges()`. It may be **private** function. Run in the EOM function. Charges are computed for excess transaction as per the charges per transaction rate. balance of account is decreased by the charges.
    10. `toString()`. Override appropriately, so that can be used for printing all details of the current bank account.
    11. `eom()`. Overrides. Called on every month end. Calls `deductCharges()`, and initializes counters that are applicable.
    12. `getAccountType()`. Overrides appropriately.

** May improve upon the client code given above, and test run all three classes **

Exercise 2

Here you create `Bank` class. A `Bank` holds collection of Bank Accounts that are mix of Saving and Current objects. Let you use the `map` container for this purpose.

The `Bank` account object should implement following operations.

1. `BankAccount open_saving_account(cust_name, ini_balance=0)`: Creates new saving bank account object. Adds to accounts collection of bank, and returns **reference** to the newly added bank account.
2. `BankAccount open_current_account(cust_name, ini_balance=0)`: Creates new current bank account object. Adds to accounts collection of bank, and returns **reference** to the newly added bank account.
3. `BankAccount find(acc_no)`: searches and returns the account object for the given account number. returned Account object can be of either type. The method throws `AccountNotFound` exception if account is not found in the bank.
4. `BankAccount close (acc_no)`: search the account object and returns. However removes the account object from the collection before returning. The method throws `AccountNotFound` exception if account is not found in the bank.
5. `print_all_accounts(std::ostream& out);` prints all accounts with their details on `ostream`.
6. You may also require creating appropriate destructor for the `Bank` class. That frees storage used by all account and emptying the maps collection.

Following is sample client code using the `Bank` class object. You may improve upon this.

```
#include <iostream>
#include "bankaccount.h"
#include "savingaccount.h"
#include "currentaccount.h"
#include "bank.h"

using namespace std;

int main() {

    Bank bank;
```

```
    bank.open_saving_account("Sunil", 2000); //1001
    bank.open_saving_account("Manisha", 1000); //1002
    bank.open_current_account("Pankaj", 1000); //1003
    bank.open_current_account("Amit", 1000); //1004

    bank.print_all_accounts( cout );

    try {
        BankAccount& ba = bank.find( 1003 );
        if (ba.getAccountType() == BankAccount::CURRENT) {
            CurrentAccount& ca = static_cast<CurrentAccount&>(ba);
            ca.setOD_Limit(5000);
        }
        ba.deposit(2000);
        ba.deposit(2500);
        ba.deposit(1000);
        ba.deposit(3000);
        ba.withdraw( 4000 );
        cout << ba.getBalance() << endl;
        ba.eom();
        cout << ba.getBalance() << endl;
        cout << ba.toString() << endl;

        bank.close(1002);

        bank.print_all_accounts( cout );

    }
    catch(AccountNotFound& e) {
        cout << "Error: " << e.what() << endl;
    }
    catch(InSufficientBalance& e) {
        cout << "Error: " << e.what() << endl;
    }
    catch(TransactionLimitExceeds& e) {
        cout << "Error: " << e.what() << endl;
    }
    catch(...) {
        cout << "Unkown Error Occurred! " << endl;
    }

    return 0;

}
```