Report on ATM System Maintenance

# Tutorial Class: 103A

|  |  |
| --- | --- |
| Student ID | Student Name |
| 20195601A | Tsoi Yiu Chik\*\*\* |
| 20088254A | Lai Jiahao |
| 20165196A | Tong Tsz Huen |
| 20093215A | Cheung Man Hei |
| 20062619A | Fung Hiu Yeung |
| 20061985A | Leung Kwan Yin |

# Class Diagrams and Assumption Setting (Before modification)

* Class diagram

Diagram, schematic

Description automatically generated

* Assumptions

1. The ATM asks the user to type an account number using the keypad, and all the output from this ATM appears on the screen.
2. The keypad only provides ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘0’, ‘.’ and ‘ENTER’ buttons.
3. Only integers are allowed to be input in the field account number, PIN, choice, and the withdrawal amount. In other words, only buttons 0-9 and ‘ENTER’ are available.
4. In other situations, such as transferring money, all buttons are available for the user to prompt.
5. The bank only plans to build one ATM system, so we need not to worry about multiple ATMs accessing the bank’s account information database at the same time.

# Class Diagrams and Explanation of Modifications

* Modified version of class diagram

A diagram of a house

Description automatically generated with medium confidence

Diagram

Description automatically generated

* Source codes

The modified parts are highlighted in yellow.

**Account:**

// Account.java

// Represents a bank account

public class Account

{

private int accountNumber; // account number

private int pin; // PIN for authentication

private double availableBalance; // funds available for withdrawal

private double totalBalance; // funds available + pending deposits

// these attributes are only for accounts having saving and chequeing purpose

private SavingAccount saving = null;

private ChequeAccount chequing = null;

// we assume existing accounts remain it original type, until the owner requests for changing

private String TYPE = "General account";

// Account constructor initializes attributes

public Account( int theAccountNumber, int thePIN,

double theAvailableBalance, double theTotalBalance )

{

accountNumber = theAccountNumber;

pin = thePIN;

availableBalance = theAvailableBalance;

totalBalance = theTotalBalance;

} // end Account constructor

// constructor initializes attributes for both saving and chequeing purpose

public Account(int theAccountNumber, int thePIN,

double savingAvailableBalance, double savingTotalBalance, double chequingAvailableBalance, double chequingTotalBalance)

{

accountNumber = theAccountNumber;

pin = thePIN;

saving = new SavingAccount(theAccountNumber, thePIN, savingAvailableBalance, savingTotalBalance);

chequing = new ChequeAccount(theAccountNumber, thePIN, chequingAvailableBalance, chequingTotalBalance);

TYPE = "Both";

}

public void \_passBalance() {

if(\_isSwapable()){

saving.credit(totalBalance);

totalBalance = 0;

}

}

// return true when the user owns both saving and chequing accounts

public boolean \_isSwapable() {

return (saving != null) && (chequing != null);

}

// swap to saving account

public SavingAccount \_swapToSaving(){

return saving;

}

// swap to chequing accout

public ChequeAccount \_swapToChequing(){

return chequing;

}

// determines whether a user-specified PIN matches PIN in Account

public boolean validatePIN( int userPIN )

{

if ( userPIN == pin )

return true;

else

return false;

} // end method validatePIN

// returns available balance

public double getAvailableBalance()

{

return availableBalance;

} // end getAvailableBalance

// returns the total balance

public double getTotalBalance()

{

return totalBalance;

} // end method getTotalBalance

// credits an amount to the account

public void credit( double amount )

{

totalBalance += amount; // add to total balance

} // end method credit

// debits an amount from the account

public void debit( double amount )

{

availableBalance -= amount; // subtract from available balance

totalBalance -= amount; // subtract from total balance

} // end method debit

// returns account number

public int getAccountNumber()

{

return accountNumber;

} // end method getAccountNumber

// returns account type

public String getType() {

return TYPE;

}

} // end class Account

**ATM:**

// ATM.java

// Represents an automated teller machine

public class ATM

{

private boolean userAuthenticated; // whether user is authenticated

private int currentAccountNumber; // current user's account number

private Screen screen; // ATM's screen

private Keypad keypad; // ATM's keypad

private CashDispenser cashDispenser; // ATM's cash dispenser

//private DepositSlot depositSlot; // ATM's deposit slot

private BankDatabase bankDatabase; // account information database

// constants corresponding to main menu options

private static final int BALANCE\_INQUIRY = 1;

private static final int WITHDRAWAL = 2;

// private static final int DEPOSIT = 3;

// added new main menu option

private static final int TRANSFER = 3;

// for exit

private static final int EXIT = 0;

// for swaping

private Account swap = null;

private static final int SWAPTOSAVING = 8;

private static final int SWAPTOCHEQUING = 9;

private static final int NOSWAPPING = 0;

// used as switch option

private final String GENERALTYPE = "General account";

private final String SAVINGTYPE = "Saving account";

private final String CHEQUEINGTYPE = "Cheque account";

private final String BOTHTYPE = "Both";

// no-argument ATM constructor initializes instance variables

public ATM()

{

userAuthenticated = false; // user is not authenticated to start

currentAccountNumber = 0; // no current account number to start

screen = new Screen(); // create screen

keypad = new Keypad(); // create keypad

cashDispenser = new CashDispenser(); // create cash dispenser

//depositSlot = new DepositSlot(); // create deposit slot

bankDatabase = new BankDatabase(); // create acct info database

} // end no-argument ATM constructor

// start ATM

public void run()

{

// welcome and authenticate user; perform transactions

while ( true )

{

// loop while user is not yet authenticated

while ( !userAuthenticated )

{

screen.displayMessageLine( "\nWelcome!" );

authenticateUser(); // authenticate user

} // end while

performTransactions(); // user is now authenticated

userAuthenticated = false; // reset before next ATM session

currentAccountNumber = 0; // reset before next ATM session

screen.displayMessageLine( "\nThank you! Goodbye!" );

} // end while

} // end method run

// attempts to authenticate user against database

private void authenticateUser()

{

screen.displayMessage( "\nPlease enter your account number: " );

int accountNumber = keypad.getInput(); // input account number

screen.displayMessage( "\nEnter your PIN: " ); // prompt for PIN

int pin = keypad.getInput(); // input PIN

// set userAuthenticated to boolean value returned by database

userAuthenticated =

bankDatabase.authenticateUser( accountNumber, pin );

// check whether authentication succeeded

if ( userAuthenticated )

{

currentAccountNumber = accountNumber; // save user's account #

if (bankDatabase.getAccountTypeString(currentAccountNumber)==BOTHTYPE) {

bankDatabase.passBalance(currentAccountNumber);

}

} // end if

else

screen.displayMessageLine(

"Invalid account number or PIN. Please try again." );

} // end method authenticateUser

// display the main menu and perform transactions

private void performTransactions(){

// Transaction currentTransaction = null;

boolean userExited = false; // user has not chosen to exit

//int mainMenuSelection = displayMainMenu();

String type = bankDatabase.getAccountTypeString(currentAccountNumber);

//int swapSelection ;

while (!userExited) {

int mainMenuSelection = displayMainMenu();

switch (mainMenuSelection) {

case SWAPTOCHEQUING:

if (swap == null) {

swap = bankDatabase.swapToChequing(currentAccountNumber);

performTransactions();

break;

}

// userExited = true;

case SWAPTOSAVING:

if(swap == null){

swap = bankDatabase.swapToSaving(currentAccountNumber);

performTransactions();

break;

}

// userExited = true;

case NOSWAPPING:

if(swap == null){

userExited = true;

break;

}

default:

// if swapped, change type to swapped account and perform related transactions.

if (swap != null) {

type = bankDatabase.getAccountTypeString(swap.getAccountNumber());

}

userExited = transactionsControl(mainMenuSelection, type);

}

mainMenuSelection = 0;

}

swap = null;

}

private boolean transactionsControl(int mainMenuSelection, String ACtype) {

Transaction currentTransaction = null;

boolean exitSignal = false;

while (!exitSignal ) {

switch ( mainMenuSelection )

{

// user chose to perform one of three transaction types

case BALANCE\_INQUIRY:

case WITHDRAWAL:

//case DEPOSIT:

case TRANSFER:

// initialize as new object of chosen type

currentTransaction =

createTransaction( mainMenuSelection, ACtype );

currentTransaction.execute(); // execute transaction when is not exit signal of general account

return exitSignal;

//break;

case EXIT: // user chose to terminate session

screen.displayMessageLine( "\nExiting the system..." );

exitSignal = true; // this ATM session should end

return exitSignal;

//break;

default: // user did not enter an integer from 1-4

screen.displayMessageLine(

"\nYou did not enter a valid selection. Try again." );

return exitSignal;

//break;

} // end switch

}

return exitSignal;

}

// display the main menu and return an input selection

private int displayMainMenu()

{

// String type = bankDatabase.getAccountTypeString(currentAccountNumber);

String type = (swap instanceof SavingAccount) ? SAVINGTYPE : (swap instanceof ChequeAccount)? CHEQUEINGTYPE:

bankDatabase.getAccountTypeString(currentAccountNumber);

swap = (swap instanceof SavingAccount)? swap = (SavingAccount) swap : (swap instanceof ChequeAccount)?

swap = (ChequeAccount) swap : swap;

int selection = 0;

// showing which type of current account

screen.displayMessageLine("\nYour current account type is: " + type);

if (bankDatabase.isSwapable(currentAccountNumber) && swap == null) {

selection = swapMenu();

}else{

selection = \_displayMainMenu();

}

return selection;

} // end method displayMainMenu

private int swapMenu() {

screen.displayMessageLine( "\nSwap menu:" );

screen.displayMessageLine( "8 - Swap to saving account" );

screen.displayMessageLine( "9 - Swap to chequing account" );

screen.displayMessageLine( "0 - Exit" );

screen.displayMessageLine("Please enter your choice: ");

return keypad.getInput();

}

// menu for every account

private int \_displayMainMenu() {

screen.displayMessageLine( "\nMain menu:" );

screen.displayMessageLine( "1 - View my balance" );

screen.displayMessageLine( "2 - Withdraw cash" );

screen.displayMessageLine( "3 - Transfer funds" );

screen.displayMessageLine( "0 - Exit\n" );

screen.displayMessageLine("Please enter your choice: ");

return keypad.getInput();

}

// return object of specified Transaction subclass

private Transaction createTransaction( int input , String ACType)

{

Transaction temp = null; // temporary Transaction variable

// determine which type of Transaction to create

if (swap != null) {

switch ( input )

{

case BALANCE\_INQUIRY: // create new BalanceInquiry transaction

temp = new BalanceInquiry(

swap, screen, bankDatabase );

break;

case WITHDRAWAL: // create new Withdrawal transaction

temp = new Withdrawal( swap, screen,

bankDatabase, keypad, cashDispenser );

break;

case TRANSFER: // create new Transfer transaction

temp = new Transfer( swap, screen,

bankDatabase, keypad);

break;

case EXIT:

break;

} // end switch

} else {

switch ( input )

{

case BALANCE\_INQUIRY: // create new BalanceInquiry transaction

temp = new BalanceInquiry(

currentAccountNumber, screen, bankDatabase );

break;

case WITHDRAWAL: // create new Withdrawal transaction

temp = new Withdrawal( currentAccountNumber, screen,

bankDatabase, keypad, cashDispenser );

break;

case TRANSFER: // create new Transfer transaction

temp = new Transfer( currentAccountNumber, screen,

bankDatabase, keypad);

break;

case EXIT:

break;

} // end switch

}

return temp; // return the newly created object

} // end method createTransaction

} // end class ATM

**ATMCaseStudy:**

// ATMCaseStudy.java

// Driver program for the ATM case study

public class ATMCaseStudy

{

// main method creates and runs the ATM

public static void main( String[] args )

{

ATM theATM = new ATM();

theATM.run();

} // end main

} // end class ATMCaseStudy

**BalanceInquiry:**

// BalanceInquiry.java

// Represents a balance inquiry ATM transaction

public class BalanceInquiry extends Transaction

{

// BalanceInquiry constructor

public BalanceInquiry( int userAccountNumber, Screen atmScreen,

BankDatabase atmBankDatabase )

{

super( userAccountNumber, atmScreen, atmBankDatabase );

} // end BalanceInquiry constructor

public BalanceInquiry( Account userAccount, Screen atmScreen,

BankDatabase atmBankDatabase )

{

super( userAccount, atmScreen, atmBankDatabase );

}

// performs the transaction

public void execute()

{

// get references to bank database and screen

BankDatabase bankDatabase = getBankDatabase();

Screen screen = getScreen();

double availableBalance,totalBalance;

//double totalBalance;

// if it is not a compound bank account, use account number on transaction

if (super.getAccount() != null) {

// get the available balance for the account involved

availableBalance =

bankDatabase.getAvailableBalance( getAccount() );

// get the total balance for the account involved

totalBalance =

bankDatabase.getTotalBalance( getAccount() );

}else{

// get the available balance for the account involved

availableBalance =

bankDatabase.getAvailableBalance( getAccountNumber() );

// get the total balance for the account involved

totalBalance =

bankDatabase.getTotalBalance( getAccountNumber() );

}

// display the balance information on the screen

screen.displayMessageLine( "\nBalance Information:" );

screen.displayMessage( " - Available balance: " );

screen.displayDollarAmount( availableBalance );

screen.displayMessage( "\n - Total balance: " );

screen.displayDollarAmount( totalBalance );

screen.displayMessageLine( "" );

} // end method execute

} // end class BalanceInquiry

**BankDatabase:**

// BankDatabase.java

// Represents the bank account information database

public class BankDatabase

{

private Account accounts[]; // array of Accounts

// no-argument BankDatabase constructor initializes accounts

public BankDatabase()

{

accounts = new Account[ 4 ]; // just 2 accounts for default testing, and 2 for account types testing

accounts[ 0 ] = new SavingAccount( 12345, 54321, 1000.0, 1200.0 );

// for users having general account, (i.e. neither saving nor chequing)

accounts[ 1 ] = new Account( 98765, 56789, 200.0, 200.0 );

// new accounts of cheque account and saving account

accounts[ 2 ] = new ChequeAccount(2, 2, 500, 1000);

// new account for users having both saving and chequing

accounts[ 3 ] = new Account(3, 3, 900000, 1000000, 200000, 400000);

} // end no-argument BankDatabase constructor

// retrieve Account object containing specified account number

private Account getAccount( int accountNumber )

{

// loop through accounts searching for matching account number

for ( Account currentAccount : accounts )

{

// return current account if match found

if ( currentAccount.getAccountNumber() == accountNumber )

return currentAccount;

} // end for

return null; // if no matching account was found, return null

} // end method getAccount

// determine whether user-specified account number and PIN match

// those of an account in the database

public boolean authenticateUser( int userAccountNumber, int userPIN )

{

// attempt to retrieve the account with the account number

Account userAccount = getAccount( userAccountNumber );

// if account exists, return result of Account method validatePIN

if ( userAccount != null )

return userAccount.validatePIN( userPIN );

else

return false; // account number not found, so return false

} // end method authenticateUser

// to check whether the account is exists in the database

public boolean accountExists(int ACnumber){

int acFound = 0;

// loop through accounts searching for matching account number

for ( Account currentAccount : accounts )

{

if(currentAccount.getAccountNumber() == ACnumber)

acFound += 1;

} // end for

// return true when only one account is matched with ACnumber

return acFound==1;

}

// return available balance of Account with specified account number

public double getAvailableBalance( int userAccountNumber )

{

return getAccount( userAccountNumber ).getAvailableBalance();

} // end method getAvailableBalance

public double getAvailableBalance( Account userAccount )

{

return userAccount.getAvailableBalance();

} // end method getAvailableBalance

// return total balance of Account with specified account number

public double getTotalBalance( int userAccountNumber )

{

return getAccount( userAccountNumber ).getTotalBalance();

} // end method getTotalBalance

// return total balance of Account with specified account number

public double getTotalBalance( Account userAccount )

{

return userAccount.getTotalBalance();

} // end method getTotalBalance

// return saving account total balance

// return chequing account total balance

public double getTotalBalance( int userAccountNumber, String ACType )

{

switch (ACType) {

case "Cheque account":

case "Saving account":

break;

default:

break;

}

return getAccount( userAccountNumber ).getTotalBalance();

} // end method getTotalBalance

public void passBalance(int bothTypeACNumber){

getAccount(bothTypeACNumber).\_passBalance();

}

// credit an amount to Account with specified account number

public void credit( int userAccountNumber, double amount )

{

getAccount( userAccountNumber ).credit( amount );

} // end method credit

// debit an amount from of Account with specified account number

public void credit( Account userAccount, double amount )

{

userAccount.credit( amount );

} // end method debit

// debit an amount from of Account with specified account number

public void debit( int userAccountNumber, double amount )

{

getAccount( userAccountNumber ).debit( amount );

} // end method debit

// debit an amount from of Account with specified account number

public void debit( Account userAccount, double amount )

{

userAccount.debit( amount );

} // end method debit

// allows backend to set cheque limit for an account

public void setChequeLimit(int userAccountNumber, double amount) {

// downcasting current account to ChequeAccount for setting limit

ChequeAccount temp = (ChequeAccount) getAccount(userAccountNumber);

temp.setLimit(amount);

}

// return user's account type in string

public String getAccountTypeString(int userAccountNumber) {

return getAccount(userAccountNumber).getType();

}

// allows backend set the interest rate for an account

public void setInterestRate(int currentAccountNumber, int rate) {

SavingAccount temp = (SavingAccount) getAccount(currentAccountNumber);

temp.setInterestRate(rate);

}

// returns current interest rate to backend

public double getCurrentRate(int currentAccountNumber) {

SavingAccount temp = (SavingAccount) getAccount(currentAccountNumber);

double rate = temp.getInterestRate();

return rate;

}

// swap to saving account

public SavingAccount swapToSaving(int currentGeneralAC) {

SavingAccount temp = getAccount(currentGeneralAC).\_swapToSaving();

return temp;

}

// swap to chequing account

public ChequeAccount swapToChequing(int currentGeneralAC) {

ChequeAccount temp = getAccount(currentGeneralAC).\_swapToChequing();

return temp;

}

// indicates whether the account is swapable

public boolean isSwapable(int currentAccountNumber) {

boolean flag = getAccount(currentAccountNumber).\_isSwapable();

return flag;

}

} // end class BankDatabase

**CashDispenser:**

// CashDispenser.java

// Represents the cash dispenser of the ATM

public class CashDispenser

{

// the default initial number of bills in the cash dispenser

private final static int INITIAL\_COUNT = 500;

private int count; // number of $20 bills remaining

// no-argument CashDispenser constructor initializes count to default

public CashDispenser()

{

count = INITIAL\_COUNT; // set count attribute to default

} // end CashDispenser constructor

// simulates dispensing of specified amount of cash

public void dispenseCash( int amount )

{

int billsRequired = amount / 100; // number of $20 bills required

count -= billsRequired; // update the count of bills

} // end method dispenseCash

// indicates whether cash dispenser can dispense desired amount

public boolean isSufficientCashAvailable( int amount )

{

int billsRequired = amount / 100; // number of $20 bills required

if ( count >= billsRequired )

return true; // enough bills available

else

return false; // not enough bills available

} // end method isSufficientCashAvailable

} // end class CashDispenser

**ChequeAccount:**

public class ChequeAccount extends Account {

private double LimitPerCheque;

private static final String TYPE = "Cheque account";

public ChequeAccount(int theAccountNumber, int thePIN,

double theAvailableBalance, double theTotalBalance) {

super(theAccountNumber, thePIN, theAvailableBalance, theTotalBalance);

LimitPerCheque = 10000;

}

// get method of this class

public void setLimit(double limit) {

LimitPerCheque = limit;

}

// returns current cheque limit

public double getLimit(){

return LimitPerCheque;

}

// returns account type

@Override

public String getType() {

return TYPE;

}

}

**Keypad:**

// Keypad.java

// Represents the keypad of the ATM

import java.util.Scanner; // program uses Scanner to obtain user input

public class Keypad

{

private Scanner input; // reads data from the command line

// no-argument constructor initializes the Scanner

public Keypad()

{

input = new Scanner( System.in );

} // end no-argument Keypad constructor

// return an integer value entered by user

public int getInput()

{

return input.nextInt(); // we assume that user enters an integer

} // end method getInput

public double getDoubleInput() {

return input.nextDouble();

}

} // end class Keypad

**SavingAccount:**

public class SavingAccount extends Account{

private double interestRate;

private static final String TYPE = "Saving account";

public SavingAccount(int theAccountNumber, int thePIN,

double theAvailableBalance, double theTotalBalance){

super(theAccountNumber, thePIN, theAvailableBalance, theTotalBalance);

interestRate = 0.001;

}

// set method of this class

public void setInterestRate(double rate ) {

interestRate = rate;

}

// get method of this class

public double getInterestRate() {

return interestRate;

}

// returns account type

@Override

public String getType() {

return TYPE;

}

}

**Screen:**

// Screen.java

// Represents the screen of the ATM

public class Screen

{

// displays a message without a carriage return

public void displayMessage( String message )

{

System.out.print( message );

} // end method displayMessage

// display a message with a carriage return

public void displayMessageLine( String message )

{

System.out.println( message );

} // end method displayMessageLine

// display a dollar amount

public void displayDollarAmount( double amount )

{

System.out.printf( "$%,.2f", amount );

} // end method displayDollarAmount

} // end class Screen

**Transaction:**

// Transaction.java

// Abstract superclass Transaction represents an ATM transaction

//import java.lang.Math;

public abstract class Transaction

{

private int accountNumber; // indicates account involved

private Screen screen; // ATM's screen

private BankDatabase bankDatabase; // account info database

private Account account = null;

// Transaction constructor invoked by subclasses using super()

public Transaction( int userAccountNumber, Screen atmScreen,

BankDatabase atmBankDatabase )

{

accountNumber = userAccountNumber;

screen = atmScreen;

bankDatabase = atmBankDatabase;

} // end Transaction constructor

// Transaction constructor invoked by subclasses using super()

public Transaction( Account userAccount, Screen atmScreen,

BankDatabase atmBankDatabase )

{

account = userAccount;

screen = atmScreen;

bankDatabase = atmBankDatabase;

} // end Transaction constructor

// return account number

public int getAccountNumber()

{

return accountNumber;

} // end method getAccountNumber

// return account

public Account getAccount(){

return account;

}// end method getAccount

// return reference to screen

public Screen getScreen()

{

return screen;

} // end method getScreen

// return reference to bank database

public BankDatabase getBankDatabase()

{

return bankDatabase;

} // end method getBankDatabase

// if it is not a compound account, return false

public boolean getFlag() {

return account != null;

}

// perform the transaction (overridden by each subclass)

abstract public void execute();

} // end class Transaction

**Transfer:**

public class Transfer extends Transaction {

private double amount; // amount to transfer

private Keypad keypad; // reference to keypad

// the target account of fund transfer

private int target;

private BankDatabase bankDatabase;

private Screen screen;

// constant corresponding to menu option to cancel

// private final static int CANCELED = 6;

// Transfer constructor

public Transfer( int userAccountNumber, Screen atmScreen,

BankDatabase atmBankDatabase, Keypad atmKeypad){

super(userAccountNumber, atmScreen, atmBankDatabase);

// initialize references to keypad

keypad = atmKeypad;

}

//end Transfer constructor

public Transfer( Account userAccount, Screen atmScreen,

BankDatabase atmBankDatabase, Keypad atmKeypad )

{

super( userAccount, atmScreen, atmBankDatabase );

keypad = atmKeypad;

}

public boolean isSufficientTransfer(double amount, double available){

return amount<=available;

}

// transfer method for non-compounded account

private void transferNormal(double availableBalance, int accountNumber, Screen screen,Keypad keypad) {

//this.keypad = keypad;

screen.displayMessage("Please enter target bank account number: ");

target = keypad.getInput();

screen.displayMessage("Please enter amount: ");

try {

amount = keypad.getDoubleInput();

if (isSufficientTransfer(amount, availableBalance) && bankDatabase.accountExists(target) && target != accountNumber) {

bankDatabase.debit(accountNumber, amount);

bankDatabase.credit(target, amount);

}else{

screen.displayMessageLine("Availavle balance is lower than transfer amount or target account unavailable.");

screen.displayMessageLine("Progress aborted.");

}

} catch (Exception e) {

// to maintain all inputs are integer, fund with cents are not considered

screen.displayMessageLine("Input mismatch! In normal mode.");

amount = 0;

}

}

private void transferCompound(double availableBalance, Account subAccount, Screen screen, Keypad keypad){

int acNum = subAccount.getAccountNumber();

// this.keypad = keypad;

screen.displayMessage("Please enter target bank account number: ");

target = keypad.getInput();

screen.displayMessage("Please enter amount: ");

try {

amount = keypad.getDoubleInput();

if (isSufficientTransfer(amount, availableBalance) && bankDatabase.accountExists(target)

&& target != acNum) {

bankDatabase.debit(subAccount, amount);

bankDatabase.credit(target, amount);

screen.displayMessageLine("Successful.");

}else{

screen.displayMessageLine("Availavle balance is lower than transfer amount or target account unavailable.");

screen.displayMessageLine("Progress aborted.");

}

} catch (Exception e) {

screen.displayMessageLine("Input mismatch! In compoud mode.");

amount = 0;

}

}

// perform transaction

public void execute()

{

// get references to bank database

bankDatabase = getBankDatabase();

boolean flag = super.getFlag();

Account subAccount = super.getAccount();

int currentAccountNumber = getAccountNumber();

double availableBalance = (subAccount != null)? bankDatabase.getAvailableBalance( getAccount()):

bankDatabase.getAvailableBalance( getAccountNumber() );

// get references to screen

screen = getScreen();

//transferCompound(availableBalance, subAccount, screen, keypad);

if(flag){

transferCompound(availableBalance, subAccount, screen, keypad);

}else{

transferNormal(availableBalance, currentAccountNumber, screen, keypad);

}

} // end method execute

}

**Withdrawl:**

// Withdrawal.java

// Represents a withdrawal ATM transaction

public class Withdrawal extends Transaction

{

private int amount; // amount to withdraw

private Keypad keypad; // reference to keypad

private CashDispenser cashDispenser; // reference to cash dispenser

// constant corresponding to menu option to customize amount

private final static int CUSTOMIZE = 6;

// constant corresponding to menu option to cancel

private final static int CANCELED = 7;

// Withdrawal constructor

public Withdrawal( int userAccountNumber, Screen atmScreen,

BankDatabase atmBankDatabase, Keypad atmKeypad,

CashDispenser atmCashDispenser )

{

// initialize superclass variables

super( userAccountNumber, atmScreen, atmBankDatabase );

// initialize references to keypad and cash dispenser

keypad = atmKeypad;

cashDispenser = atmCashDispenser;

} // end Withdrawal constructor

public Withdrawal( Account userAccount, Screen atmScreen,

BankDatabase atmBankDatabase,Keypad atmKeypad,

CashDispenser atmCashDispenser )

{

super( userAccount, atmScreen, atmBankDatabase );

keypad = atmKeypad;

cashDispenser = atmCashDispenser;

}

// perform transaction

public void execute()

{

boolean DisableDispenser = false; // cash was not dispensed yet

// get references to bank database and screen

BankDatabase bankDatabase = getBankDatabase();

Screen screen = getScreen();

Account subAccount = super.getAccount();

double availableBalance = (subAccount != null)? bankDatabase.getAvailableBalance( getAccount()):

bankDatabase.getAvailableBalance( getAccountNumber() ); // amount available for withdrawal

boolean flag = super.getFlag();

int currentAccountNumber = getAccountNumber();

do{

if (flag) {

DisableDispenser = compoundWithdraw(availableBalance, subAccount, screen, keypad, bankDatabase, DisableDispenser);

}else{

DisableDispenser = normalWithdraw(availableBalance, currentAccountNumber, screen, keypad, bankDatabase, DisableDispenser);

}

}while(!DisableDispenser);

// loop until cash is dispensed or the user cancels

} // end method execute

private boolean normalWithdraw(double availableBalance, int currentAccountNumber, Screen screen,

Keypad keypad\_Keypad, BankDatabase database, boolean disabled) {

// obtain a chosen withdrawal amount from the user

amount = displayMenuOfAmounts();

// check whether user chose a withdrawal amount or canceled

if ( amount != CANCELED )

{

// check whether the user has enough money in the account

if ( amount <= availableBalance )

{

// check whether the cash dispenser has enough money

if ( cashDispenser.isSufficientCashAvailable( amount ) )

{

// update the account involved to reflect withdrawal

database.debit( getAccountNumber(), amount );

cashDispenser.dispenseCash( amount ); // dispense cash

disabled = true; // cash was dispensed

// instruct user to take cash

screen.displayMessageLine(

"\nPlease take your cash now." );

} // end if

else // cash dispenser does not have enough cash

screen.displayMessageLine(

"\nInsufficient cash available in the ATM." +

"\n\nPlease choose a smaller amount." );

} // end if

else // not enough money available in user's account

{

screen.displayMessageLine(

"\nInsufficient funds in your account." +

"\n\nPlease choose a smaller amount." );

} // end else

} // end if

else // user chose cancel menu option

{

screen.displayMessageLine( "\nCanceling transaction..." );

disabled = true; // return to main menu because user canceled

} // end else

return disabled;

}

private boolean compoundWithdraw(double availableBalance, Account subAccount, Screen screen, Keypad keypad\_Keypad,

BankDatabase database, boolean disabled) {

// obtain a chosen withdrawal amount from the user

amount = displayMenuOfAmounts();

// check whether user chose a withdrawal amount or canceled

if ( amount != CANCELED )

{

// check whether the user has enough money in the account

if ( amount <= availableBalance )

{

// check whether the cash dispenser has enough money

if ( cashDispenser.isSufficientCashAvailable( amount ) )

{

// update the account involved to reflect withdrawal

database.debit( subAccount, amount );

cashDispenser.dispenseCash( amount ); // dispense cash

disabled = true; // cash was dispensed

// instruct user to take cash

screen.displayMessageLine(

"\nPlease take your cash now." );

} // end if

else // cash dispenser does not have enough cash

screen.displayMessageLine(

"\nInsufficient cash available in the ATM." +

"\n\nPlease choose a smaller amount." );

} // end if

else // not enough money available in user's account

{

screen.displayMessageLine(

"\nInsufficient funds in your account." +

"\n\nPlease choose a smaller amount." );

} // end else

} // end if

else // user chose cancel menu option

{

screen.displayMessageLine( "\nCanceling transaction..." );

disabled = true; // return to main menu because user canceled

} // end else

return disabled;

}

// display a menu of withdrawal amounts and the option to cancel;

// return the chosen amount or 0 if the user chooses to cancel

private int displayMenuOfAmounts()

{

int userChoice = 0; // local variable to store return value

Screen screen = getScreen(); // get screen reference

// array of amounts to correspond to menu numbers

int amounts[] = { 0, 200, 400, 600, 800, 1000};

// loop while no valid choice has been made

while ( userChoice == 0 )

{

// display the menu

screen.displayMessageLine( "\nWithdrawal Menu:" );

screen.displayMessageLine( "1 - $200" );

screen.displayMessageLine( "2 - $400" );

screen.displayMessageLine( "3 - $600" );

screen.displayMessageLine( "4 - $800" );

screen.displayMessageLine( "5 - $1000" );

screen.displayMessageLine( "6 - Custom amount" );

screen.displayMessageLine( "7 - Cancel transaction" );

screen.displayMessage( "\nChoose a withdrawal amount: " );

int input = keypad.getInput(); // get user input through keypad

// determine how to proceed based on the input value

switch ( input )

{

case 1: // if the user chose a withdrawal amount

case 2: // (i.e., chose option 1, 2, 3, 4 or 5), return the

case 3: // corresponding amount from amounts array

case 4:

case 5:

userChoice = amounts[ input ]; // save user's choice

break;

case CUSTOMIZE: // get user input

screen.displayMessageLine("Please enter withdrawl option: ");

int temp = keypad.getInput();

if (checkIsMultiple(temp, screen)) {

userChoice = temp;

}

break;

case CANCELED: // the user chose to cancel

userChoice = CANCELED; // save user's choice

break;

default: // the user did not enter a value from 1-6

screen.displayMessageLine(

"\nIvalid selection. Try again." );

} // end switch

} // end while

return userChoice; // return withdrawal amount or CANCELED

} // end method displayMenuOfAmounts

// to check whether user's input is a multiple of 100

public boolean checkIsMultiple(int input, Screen screen){

int mod = input % 100;

if (mod == 0) {

return true;

}else{

screen.displayMessageLine("Input is not a multiple of $100, aborting...");

return false;

}

}

} // end class Withdrawal

* Explanations of the key program statements

Part (A):

Before modification:

// array of amounts to correspond to menu numbers

int amounts[] = { 0, 20, 40, 60, 100, 200 };

// loop while no valid choice has been made

while ( userChoice == 0 )

{

// display the menu

screen.displayMessageLine( "\nWithdrawal Menu:" );

screen.displayMessageLine( "1 - $20" );

screen.displayMessageLine( "2 - $40" );

screen.displayMessageLine( "3 - $60" );

screen.displayMessageLine( "4 - $100" );

screen.displayMessageLine( "5 - $200" );

screen.displayMessageLine( "6 - Cancel transaction" );

screen.displayMessage( "\nChoose a withdrawal amount: " );

After modification:

// array of amounts to correspond to menu numbers

int amounts[] = { 0, 200, 400, 600, 800, 1000};

// loop while no valid choice has been made

while ( userChoice == 0 )

{

// display the menu

screen.displayMessageLine( "\nWithdrawal Menu:" );

screen.displayMessageLine( "1 - $200" );

screen.displayMessageLine( "2 - $400" );

screen.displayMessageLine( "3 - $600" );

screen.displayMessageLine( "4 - $800" );

screen.displayMessageLine( "5 - $1000" );

screen.displayMessageLine( "6 - Custom amount" );

screen.displayMessageLine( "7 - Cancel transaction" );

screen.displayMessage( "\nChoose a withdrawal amount: " );

The prototype was originally designed for the use in USA. To fit in the situation in HK, we have adjusted the options for cash withdrawal by providing only the multiples HKD100, HKD500, or HKD1000, which in our ATM is $200, $400, $600, $800, and $1000. We have also added an option for the user to key in the amounts by themselves, which the amount was not shown in the standard withdrawal amounts, for example $1100.

Part (B):

A new account saving account was implemented in our modified ATM system.

public class SavingAccount extends Account{

private double interestRate;

private static final String TYPE = "Saving account";

public SavingAccount(int theAccountNumber, int thePIN,

double theAvailableBalance, double theTotalBalance){

super(theAccountNumber, thePIN, theAvailableBalance, theTotalBalance);

interestRate = 0.001;

}

// set method of this class

public void setInterestRate(double rate) {

interestRate = rate;

}

// get method of this class

public double getInterestRate() {

return interestRate;

}

// returns account type

@Override

public String getType() {

return TYPE;

}

}

A new subclass SavingAccount which extends from class Account was added to our ATM system. It has a special attribute called the interest rate (interestRate) with default value of 0.1% per annum in our system.

A new account cheque account was implemented in out modified ATM system.

public class ChequeAccount extends Account {

private double LimitPerCheque;

private static final String TYPE = "Cheque account";

public ChequeAccount(int theAccountNumber, int thePIN,

double theAvailableBalance, double theTotalBalance) {

super(theAccountNumber, thePIN, theAvailableBalance, theTotalBalance);

LimitPerCheque = 10000;

}

// get method of this class

public void setLimit(double limit) {

LimitPerCheque = limit;

}

// returns current cheque limit

public double getLimit(){

return LimitPerCheque;

}

// returns account type

@Override

public String getType() {

return TYPE;

}

}

A new subclass ChequeAccount which extends from class Account was added to our ATM system. It has a special attribute called the limit per cheque (LimitPerCheque) with default value of $10000 in this account.

Part (C):

Deposit function was removed in our modified ATM system.

Part (D):

A new function transfer was implemented in our modified ATM system.

public class Transfer extends Transaction {

private double amount; // amount to transfer

private Keypad keypad; // reference to keypad

// the target account of fund transfer

private int target;

private BankDatabase bankDatabase;

private Screen screen;

// constant corresponding to menu option to cancel

// private final static int CANCELED = 6;

// Transfer constructor

public Transfer( int userAccountNumber, Screen atmScreen,

BankDatabase atmBankDatabase, Keypad atmKeypad){

super(userAccountNumber, atmScreen, atmBankDatabase);

// initialize references to keypad

keypad = atmKeypad;

}

//end Transfer constructor

public Transfer( Account userAccount, Screen atmScreen,

BankDatabase atmBankDatabase, Keypad atmKeypad )

{

super( userAccount, atmScreen, atmBankDatabase );

keypad = atmKeypad;

}

public boolean isSufficientTransfer(double amount, double available){

return amount<=available;

}

// transfer method for non-compounded account

private void transferNormal(double availableBalance, int accountNumber, Screen screen,Keypad keypad) {

//this.keypad = keypad;

screen.displayMessage("Please enter target bank account number: ");

target = keypad.getInput();

screen.displayMessage("Please enter amount: ");

try {

amount = keypad.getDoubleInput();

if (isSufficientTransfer(amount, availableBalance) && bankDatabase.accountExists(target) && target != accountNumber) {

bankDatabase.debit(accountNumber, amount);

bankDatabase.credit(target, amount);

}else{

screen.displayMessageLine("Availavle balance is lower than transfer amount or target account unavailable.");

screen.displayMessageLine("Progress aborted.");

}

} catch (Exception e) {

// to maintain all inputs are integer, fund with cents are not considered

screen.displayMessageLine("Input mismatch! In normal mode.");

amount = 0;

}

}

private void transferCompound(double availableBalance, Account subAccount, Screen screen, Keypad keypad){

int acNum = subAccount.getAccountNumber();

// this.keypad = keypad;

screen.displayMessage("Please enter target bank account number: ");

target = keypad.getInput();

screen.displayMessage("Please enter amount: ");

try {

amount = keypad.getDoubleInput();

if (isSufficientTransfer(amount, availableBalance) && bankDatabase.accountExists(target)

&& target != acNum) {

bankDatabase.debit(subAccount, amount);

bankDatabase.credit(target, amount);

screen.displayMessageLine("Successful.");

}else{

screen.displayMessageLine("Availavle balance is lower than transfer amount or target account unavailable.");

screen.displayMessageLine("Progress aborted.");

}

} catch (Exception e) {

screen.displayMessageLine("Input mismatch! In compoud mode.");

amount = 0;

}

}

// perform transaction

public void execute()

{

// get references to bank database

bankDatabase = getBankDatabase();

boolean flag = super.getFlag();

Account subAccount = super.getAccount();

int currentAccountNumber = getAccountNumber();

double availableBalance = (subAccount != null)? bankDatabase.getAvailableBalance( getAccount()):

bankDatabase.getAvailableBalance( getAccountNumber() );

// get references to screen

screen = getScreen();

//transferCompound(availableBalance, subAccount, screen, keypad);

if(flag){

transferCompound(availableBalance, subAccount, screen, keypad);

}else{

transferNormal(availableBalance, currentAccountNumber, screen, keypad);

}

} // end method execute

}

A new subclass Transfer which extends from class Transaction was added to our ATM system, for transferring fund from one bank account to another bank account.

# Further addition on code

Other than the modifications above, we have also optimized our code to perform those adjustments. Including new accounts, menus, swapping memory and extra transaction control mechanism.

## New accounts:

Four new accounts with distinct properties were created for testing in class BankDatabase.

public BankDatabase()

{

accounts = new Account[ 4 ]; // just 2 accounts for default testing, and 2 for account types testing

accounts[ 0 ] = new SavingAccount( 12345, 54321, 1000.0, 1200.0 );

// for users having general account, (i.e. neither saving nor chequing)

accounts[ 1 ] = new Account( 98765, 56789, 200.0, 200.0 );

// new accounts of cheque account and saving account

accounts[ 2 ] = new ChequeAccount(2, 2, 500, 1000);

// new account for users having both saving and chequing

accounts[ 3 ] = new Account(3, 3, 900000, 1000000, 200000, 400000);

} // end no-argument BankDatabase constructor

Instead of having only saving and cheque accounts, general and compound accounts were implemented in our ATM system. Customers can decide on whether to request their account to be a compound account or not, which means both saving and cheque account. General and compound accounts are using constructors with different signatures.

If customers request to have a compound account, an object of Account will be generated with 2 subaccounts:

// constructor initializes attributes for both saving and chequeing purpose

public Account(int theAccountNumber, int thePIN,

double savingAvailableBalance, double savingTotalBalance, double chequingAvailableBalance, double chequingTotalBalance)

{

accountNumber = theAccountNumber;

pin = thePIN;

saving = new SavingAccount(theAccountNumber, thePIN, savingAvailableBalance, savingTotalBalance);

chequing = new ChequeAccount(theAccountNumber, thePIN, chequingAvailableBalance, chequingTotalBalance);

TYPE = "Both";

}

For processing the transaction of a compound account, we have added different versions of credit and debit function. In other words, we must swap to a subaccount when the user requests. Therefore, a *bankdatabase.getAccount()* has developed to return a subaccount object for further operation.

It starts from the main class, the ATM.java. An instance variable of Account was created, namely **swap**. It is for storing subaccount from compound account temporary.

// for swaping

private Account swap = null;

Once the user entered a compound account, *performTransactions()* will be executed. It first retrieves current account’s type by using *bankDatabase.getAccountTypeString(currentAccountNumber)*.

// display the main menu and perform transactions

private void performTransactions(){

// Transaction currentTransaction = null;

boolean userExited = false; // user has not chosen to exit

//int mainMenuSelection = displayMainMenu();

String type = bankDatabase.getAccountTypeString(currentAccountNumber);

//int swapSelection ;

Then, a *swapMenu()* will be displayed for user to switch to their subaccount. It is because the current one is still not a subaccount and it meets the conditions of bankDatabase.isSwapable(currentAccountNumber) && swap == null in *displayMainMenu()*.

private int swapMenu() {

screen.displayMessageLine( "\nSwap menu:" );

screen.displayMessageLine( "8 - Swap to saving account" );

screen.displayMessageLine( "9 - Swap to chequing account" );

screen.displayMessageLine( "0 - Exit" );

screen.displayMessageLine("Please enter your choice: ");

return keypad.getInput();

}

After selection, a switch case has entered.

while (!userExited) {

int mainMenuSelection = displayMainMenu();

switch (mainMenuSelection) {

case SWAPTOCHEQUING:

if (swap == null) {

swap = bankDatabase.swapToChequing(currentAccountNumber);

performTransactions();

break;

}

// userExited = true;

case SWAPTOSAVING:

if(swap == null){

swap = bankDatabase.swapToSaving(currentAccountNumber);

performTransactions();

break;

}

// userExited = true;

case NOSWAPPING:

if(swap == null){

userExited = true;

break;

}

default:

// if swapped, change type to swapped account and perform related transactions.

if (swap != null) {

type = bankDatabase.getAccountTypeString(swap.getAccountNumber());

}

userExited = transactionsControl(mainMenuSelection, type);

}

mainMenuSelection = 0;

}

swap = null;

}

For both cases **SWAPTOCHEQUING** and **SWAPTOSAVING**, if **swap** has nothing, the *bankDatabase.swapToChequing(currentAccountNumber)* or *bankDatabase.swapToSaving(currentAccountNumber)* will be executed and it returns a cheque or saving account with the current user number. To continue, we applied recursion in this. While an account object has passed to **swap**, *performTransactions()* executes again. But, this time, **swap** != null, hence down-casting will be performed for **swap** and **selection** will be the return of *\_displayMainMenu()*. When the user finishes choosing the option from the menu, the selection in the inner *performTransactions()* must be in the case default, therefore *transactionsControl(mainMenuSelection, type)* executes with Boolean return for **userExited**.

In *transactionsControl(mainMenuSelection, type)*, local variable **currentTransaction** receives the return from *createTransaction( mainMenuSelection, ACtype )* and executes it, whereas the **mainMenuSelection** is not equal to **EXIT***.* After execution, it usually returns **exitSignal** (false) unless the user choose to exit it.

private boolean transactionsControl(int mainMenuSelection, String ACtype) {

Transaction currentTransaction = null;

boolean exitSignal = false;

while (!exitSignal ) {

switch ( mainMenuSelection )

{

// user chose to perform one of three transaction types

case BALANCE\_INQUIRY:

case WITHDRAWAL:

//case DEPOSIT:

case TRANSFER:

// initialize as new object of chosen type

currentTransaction =

createTransaction( mainMenuSelection, ACtype );

currentTransaction.execute(); // execute transaction when is not exit signal of general account

return exitSignal;

//break;

case EXIT: // user chose to terminate session

screen.displayMessageLine( "\nExiting the system..." );

exitSignal = true; // this ATM session should end

return exitSignal;

//break;

default: // user did not enter an integer from 1-4

screen.displayMessageLine(

"\nYou did not enter a valid selection. Try again." );

return exitSignal;

//break;

} // end switch

}

return exitSignal;

}

In *createTransaction( mainMenuSelection, ACtype )*, there is an if-condition for determining what kind of transaction it should creat. If **swap** is not null, i.e. the user has entered a compound account, this method creates and returns transaction **temp** using **swap** as one of the constructor arguments.

// return object of specified Transaction subclass

private Transaction createTransaction( int input , String ACType)

{

Transaction temp = null; // temporary Transaction variable

// determine which type of Transaction to create

if (swap != null) {

switch ( input )

{

case BALANCE\_INQUIRY: // create new BalanceInquiry transaction

temp = new BalanceInquiry(

swap, screen, bankDatabase );

break;

case WITHDRAWAL: // create new Withdrawal transaction

temp = new Withdrawal( swap, screen,

bankDatabase, keypad, cashDispenser );

break;

case TRANSFER: // create new Transfer transaction

temp = new Transfer( swap, screen,

bankDatabase, keypad);

break;

case EXIT:

break;

} // end switch

} else {

switch ( input )

{

case BALANCE\_INQUIRY: // create new BalanceInquiry transaction

temp = new BalanceInquiry(

currentAccountNumber, screen, bankDatabase );

break;

case WITHDRAWAL: // create new Withdrawal transaction

temp = new Withdrawal( currentAccountNumber, screen,

bankDatabase, keypad, cashDispenser );

break;

case TRANSFER: // create new Transfer transaction

temp = new Transfer( currentAccountNumber, screen,

bankDatabase, keypad);

break;

case EXIT:

break;

} // end switch

}

return temp; // return the newly created object

} // end method createTransaction

Therefore, overloading method in other class like BankDatabase.java is a vital strategy for this. For example, *BankDatabase.credit(Account userAccount, double amount)* is overloading the original one, for the usage of the compound account. If we apply the original one, *getAccount(userAccountNumber)* will always returns an Account object but not a subaccount, so we are required to pass a subaccount object to the method directly.

If **swap** is null, i.e. user is not entered a compound account, *createTransaction( mainMenuSelection, ACtype**)* will create and return transaction using original set of sub-transactions and return normally.

In Transaction.java, a private instance variable in Account is added and is initialized to null.

private Account account = null;

This step offers memories for compound account. As a superclass of other transactions, the constructor is also overloaded to adapt different situation, which is about the compound account. In the middle of file, a method **getAccount()** is designed for every subclass of getting **swap** account on performing transactions.

// return account

public Account getAccount(){

return account;

}// end method getAccount

In every sub-transaction, *super.getAccount()* is used to retrieve a subaccount for further transaction.

For BalanceInquiry.java, if a subaccount is loaded into the super transaction, *getAvailableBalance( Account )* and *getTotalBalance( Account )* will be executed to retrieve the desired values.

if (super.getAccount() != null) {

// get the available balance for the account involved

availableBalance =

bankDatabase.getAvailableBalance( getAccount() );

For Withdrawal.java , it is basically as same as Transfer.java, getting **flag** and **subAccount** for the if-condition to determine which withdraw method should apply, namely *compoundWithdraw()* and *compoundWithdraw()*. In *compoundWithdraw()*, it accepts a parameter in Account type while *normalWithdraw()* accepts account number as the parameter.

private boolean compoundWithdraw(double availableBalance, Account subAccount, Screen screen, Keypad keypad\_Keypad,

BankDatabase database, boolean disabled) {

// obtain a chosen withdrawal amount from the user

amount = displayMenuOfAmounts();

// check whether user chose a withdrawal amount or canceled

if ( amount != CANCELED )

{

// check whether the user has enough money in the account

if ( amount <= availableBalance )

{

// check whether the cash dispenser has enough money

if ( cashDispenser.isSufficientCashAvailable( amount ) )

{

// update the account involved to reflect withdrawal

database.debit( subAccount, amount );

cashDispenser.dispenseCash( amount ); // dispense cash

disabled = true; // cash was dispensed

// instruct user to take cash

screen.displayMessageLine(

"\nPlease take your cash now." );

} // end if

else // cash dispenser does not have enough cash

screen.displayMessageLine(

"\nInsufficient cash available in the ATM." +

"\n\nPlease choose a smaller amount." );

} // end if

else // not enough money available in user's account

{

screen.displayMessageLine(

"\nInsufficient funds in your account." +

"\n\nPlease choose a smaller amount." );

} // end else

} // end if

else // user chose cancel menu option

{

screen.displayMessageLine( "\nCanceling transaction..." );

disabled = true; // return to main menu because user canceled

} // end else

return disabled;

}

For Transfer.java, *super.getFlag()* is used to check whether there is a subaccount for running transfer transaction.

boolean flag = super.getFlag();

After that, it loads the subaccount to a local variable **subaccount** whatever it is, e.g. null. Following up is an if-condition, it guides program whether *transferCompound()* should perform or not, if not(i.e., not a compound account), executes *transferNormal()* instead.

if(flag){

transferCompound(availableBalance, subAccount, screen, keypad);

}else{

transferNormal(availableBalance, currentAccountNumber, screen, keypad);

}

In addition, all fund transfer to compound account will be saved in **totalBalance** in Account class temporarily. After *authenticateUser()* the received balance will be passed to saving, the subaccount in SavingAccount type, by *bankDatabase.passBalance(currentAccountNumber).*

// attempts to authenticate user against database

private void authenticateUser()

{

screen.displayMessage( "\nPlease enter your account number: " );

int accountNumber = keypad.getInput(); // input account number

screen.displayMessage( "\nEnter your PIN: " ); // prompt for PIN

int pin = keypad.getInput(); // input PIN

// set userAuthenticated to boolean value returned by database

userAuthenticated =

bankDatabase.authenticateUser( accountNumber, pin );

// check whether authentication succeeded

if ( userAuthenticated )

{

currentAccountNumber = accountNumber; // save user's account #

if (bankDatabase.getAccountTypeString(currentAccountNumber)==BOTHTYPE) {

bankDatabase.passBalance(currentAccountNumber);

}

} // end if

else

screen.displayMessageLine(

"Invalid account number or PIN. Please try again." );

} // end method authenticateUser

After all transactions in current subaccount has been completed, i.e. user entered 0 as main menu choice, the inner *performTransactions()* ends. At this moment, Boolean variable **userExited** in outer *performTransactions()* is still false. Therefore, one more while loop will go on to perform *swapMenu()* again as swap will become null in the first exit from main menu.

while (!userExited) {

int mainMenuSelection = displayMainMenu();

switch (mainMenuSelection) {

case SWAPTOCHEQUING:

if (swap == null) {

swap = bankDatabase.swapToChequing(currentAccountNumber);

performTransactions();

break;

}

// userExited = true;

case SWAPTOSAVING:

if(swap == null){

swap = bankDatabase.swapToSaving(currentAccountNumber);

performTransactions();

break;

}

// userExited = true;

case NOSWAPPING:

if(swap == null){

userExited = true;

break;

}

default:

// if swapped, change type to swapped account and perform related transactions.

if (swap != null) {

type = bankDatabase.getAccountTypeString(swap.getAccountNumber());

}

userExited = transactionsControl(mainMenuSelection, type);

}

mainMenuSelection = 0;

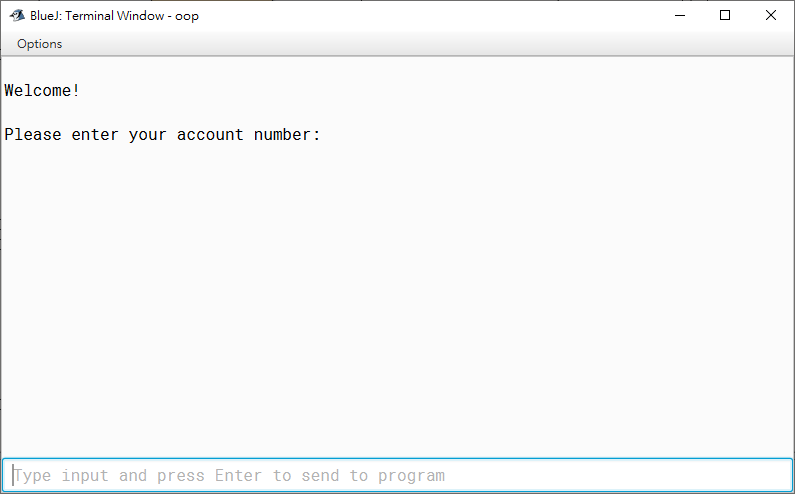
}

swap = null;

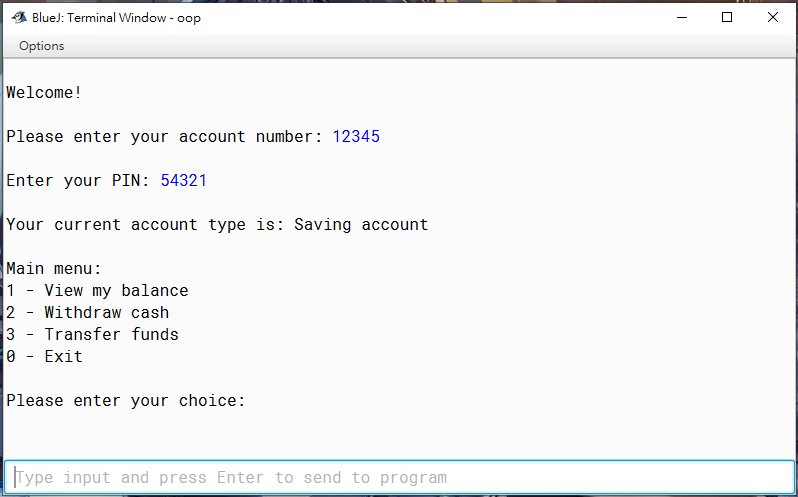
}

When user confirms exit the ATM, the while loop in outer *performTransactions()* ends.

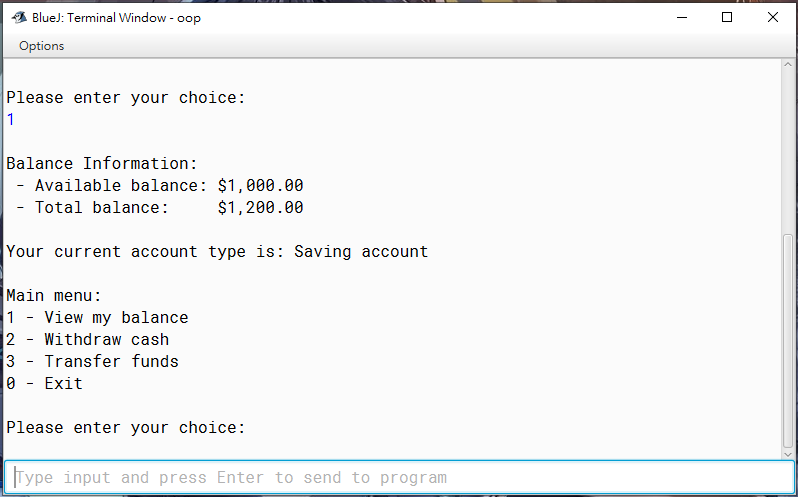
* Test cases



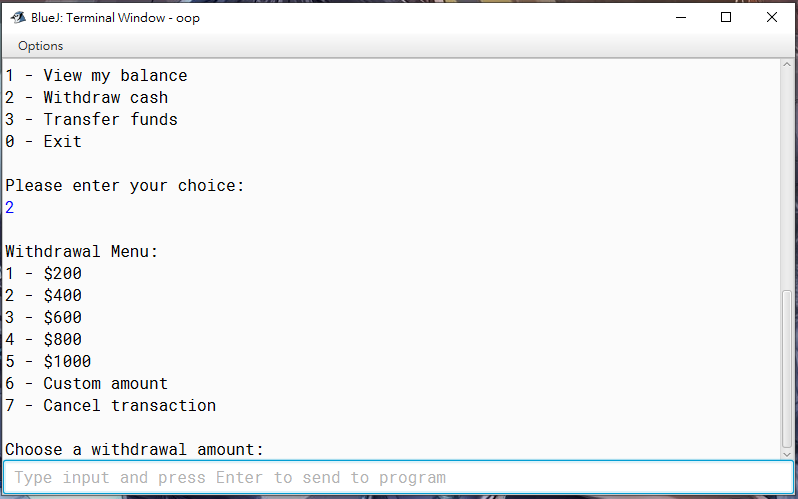
This is the interface of our program.



Account 12345 with pin 54321.

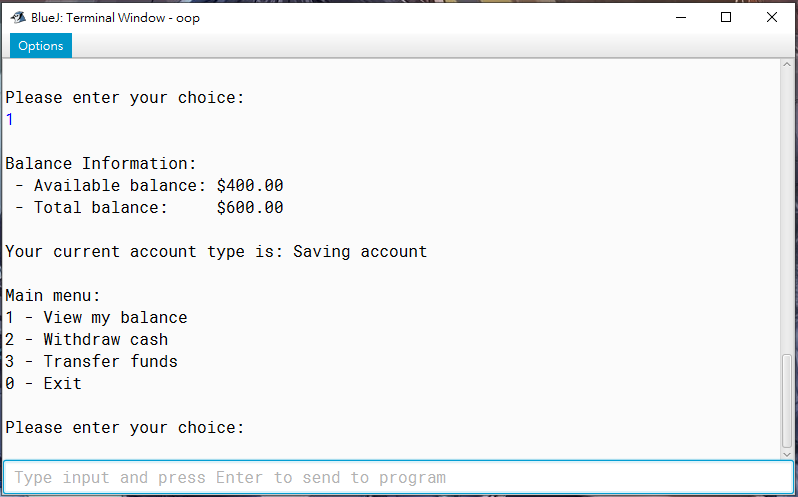


Input 1 to display the balance information.

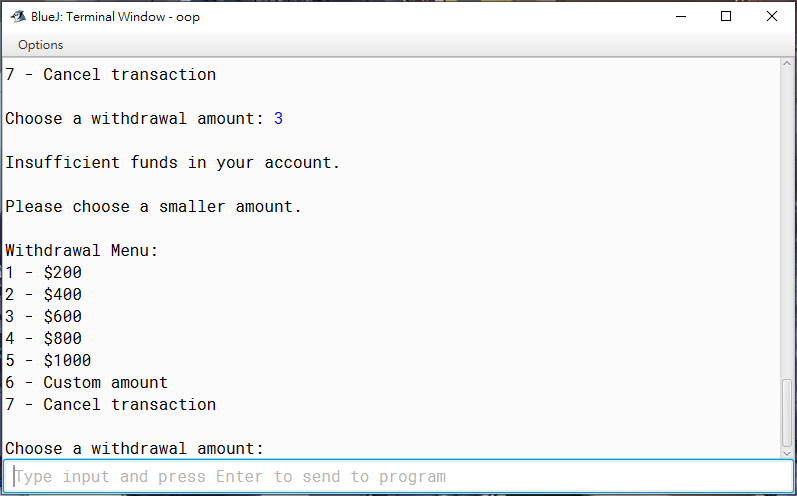


Input 2 to withdraw cash from the account. There are 5 withdrawal options with specified amount and an option for user to input custom amount.

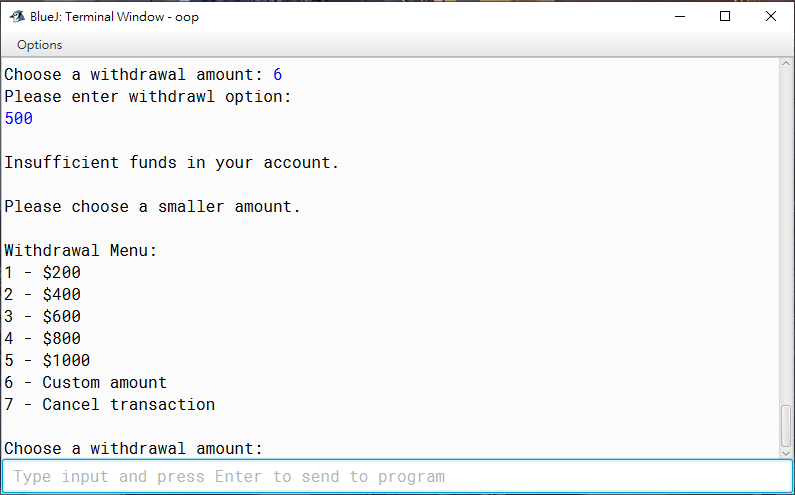




Show case that we have inputted 3 to withdraw $600 from account. The available balance has lowered to $400 from $1000 and the total balance has lowered to $600 from $1200.



If we input 3 again, “Insufficient funds in your account.” shows because there is no enough available balance left in the account.



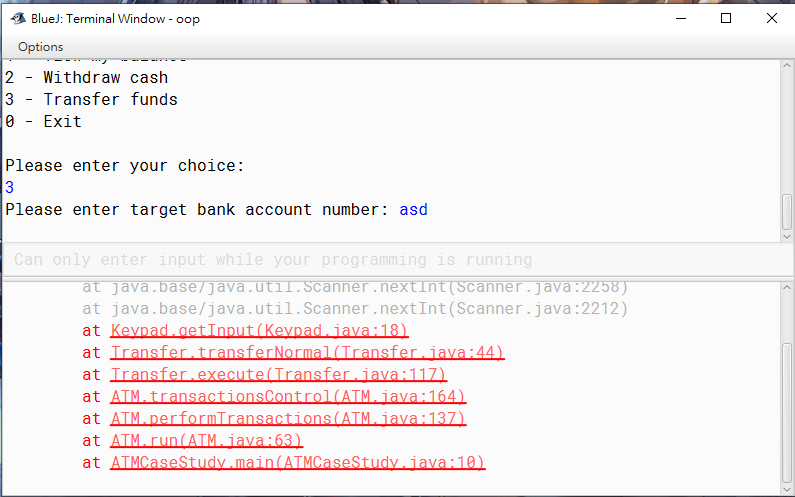
Input custom amount $500 will also show the same result.



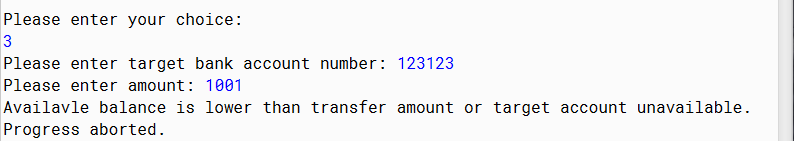
When unexpected input inputted, it displays “Invalid selection. Try again”.

# 

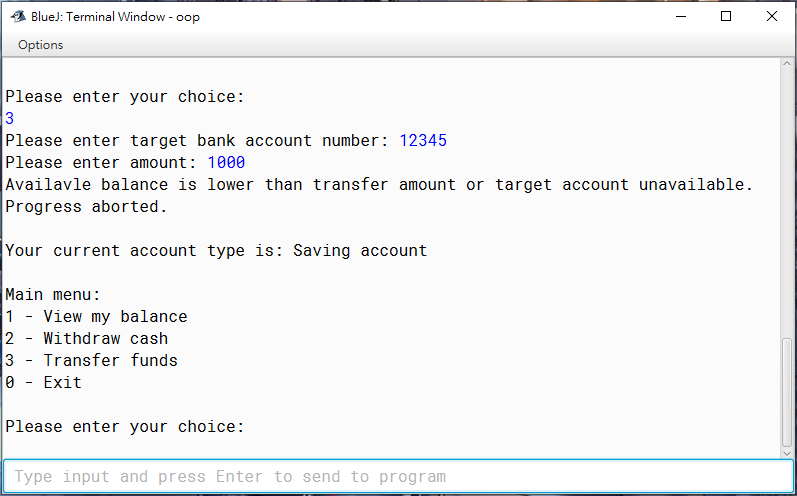
Custom amount can’t be a double value.



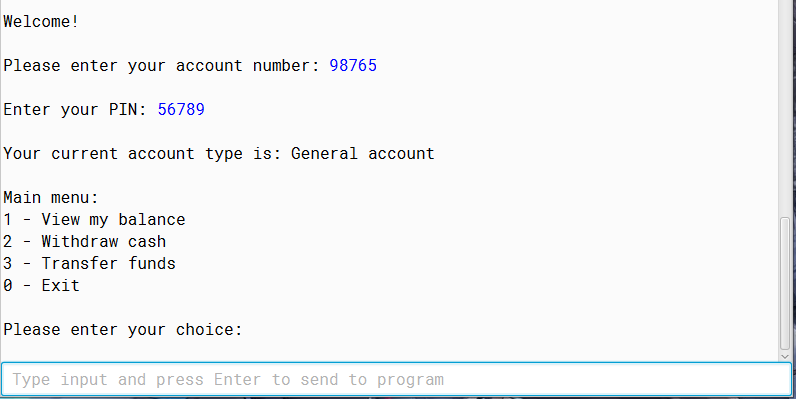
Showing the transfer funds function. Input can’t be value other than numbers.

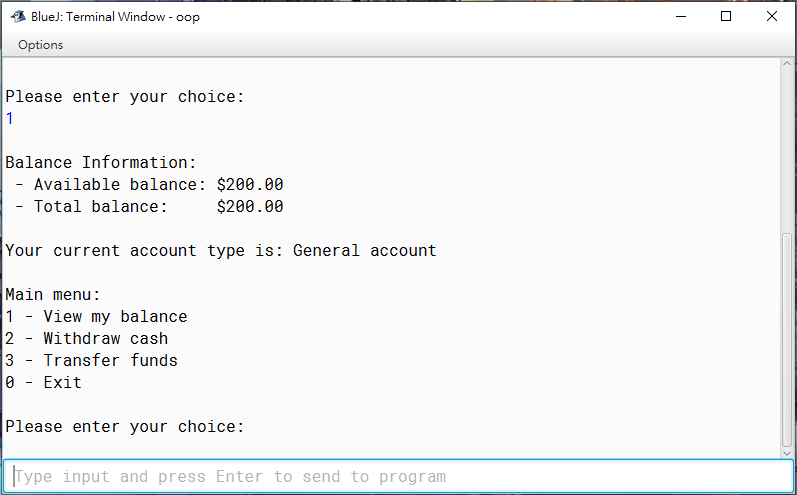


When inputting not exist account and amount above the account available balance, “Available balance is lower than transfer amount or target account unavailable. Progress aborted.” shows.

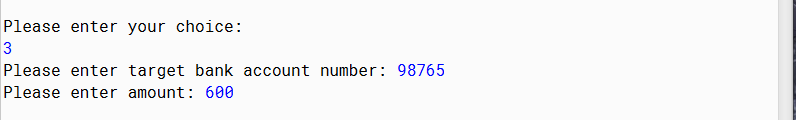


Users can’t transfer funds to the exact same account.

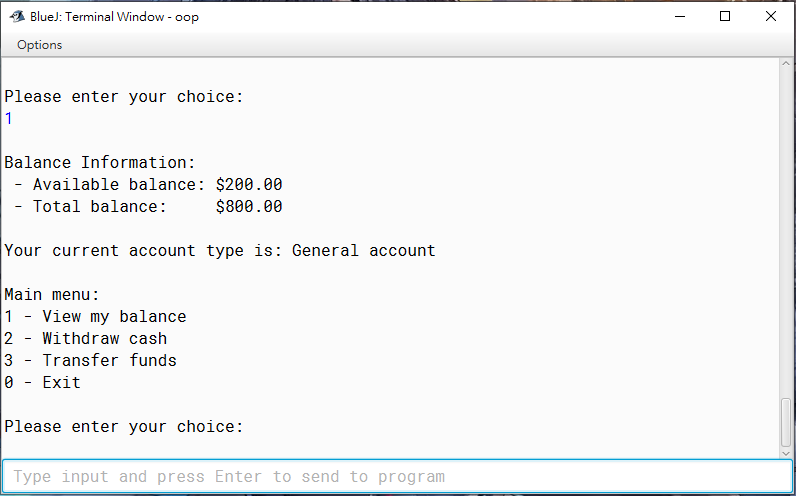




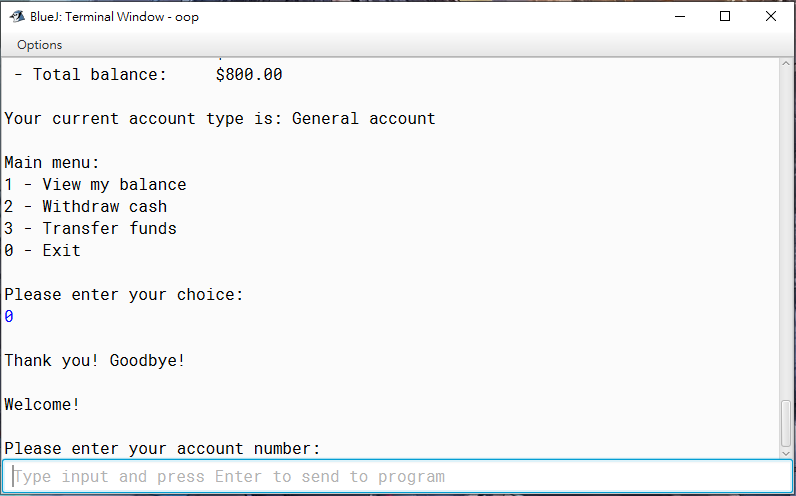
Original status of account 98765.



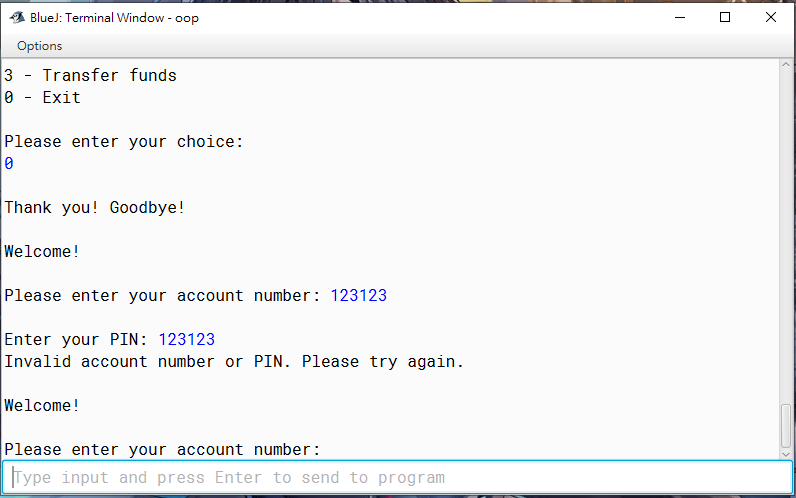
Transfer $600 from account 12345 to account 98765.



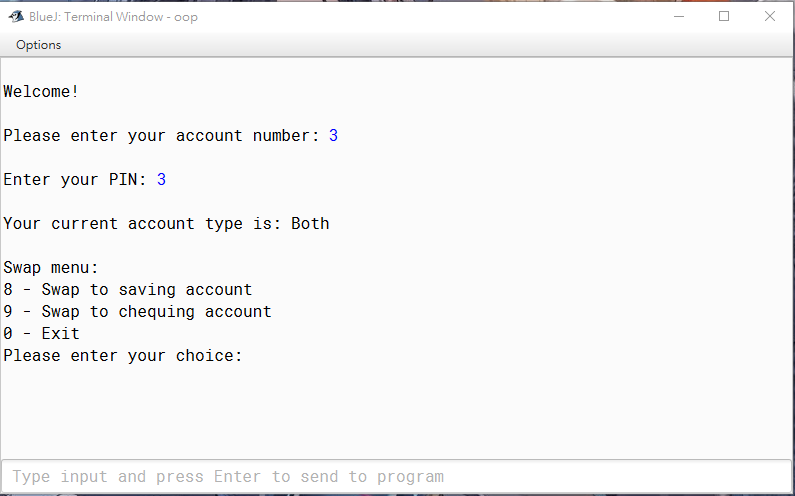
The total balance of account 98765 is added $600.



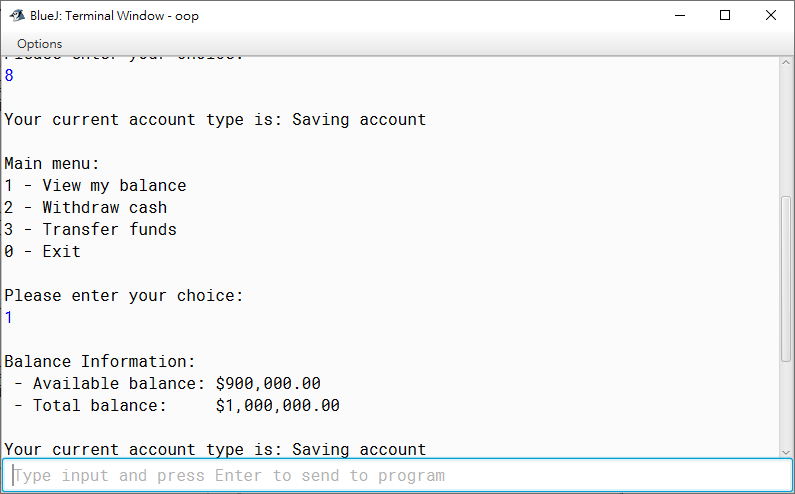
Input 0 can logout from your account and login to another one.



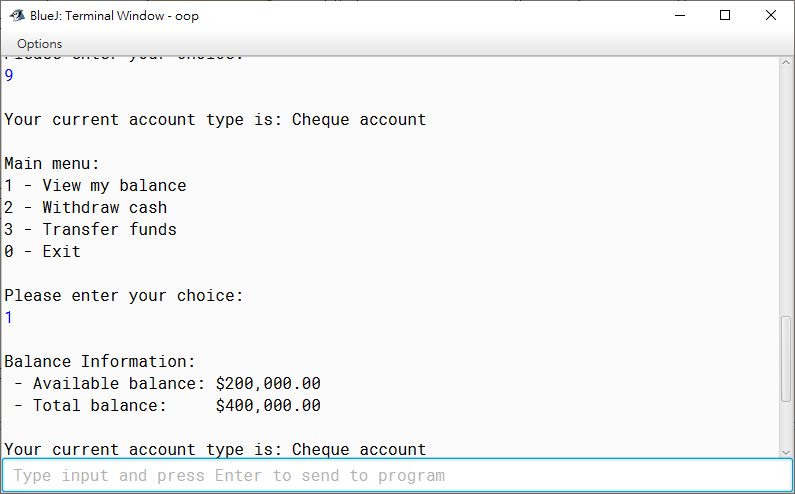
When inputting account that does not exist, the program will let you input again.



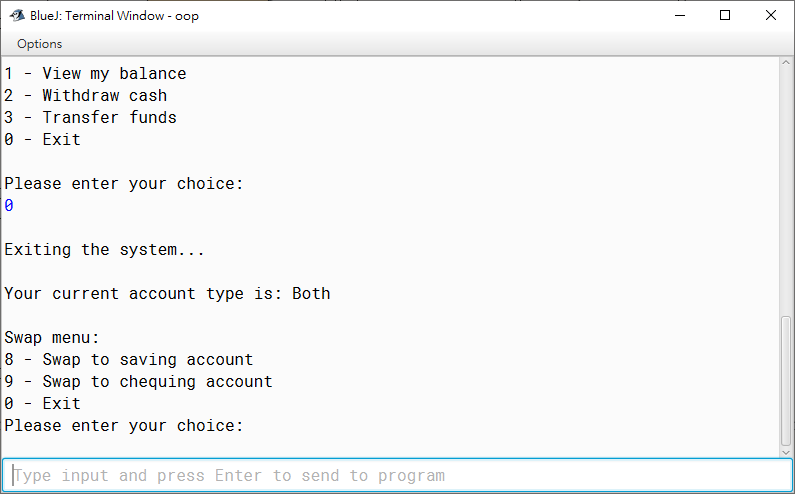
Account 3 is a compound account which consist of saving and cheque account.



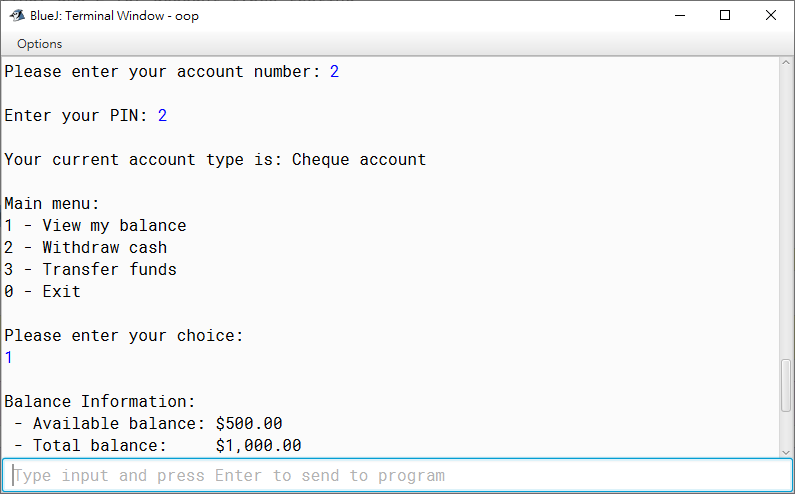
This is the saving account of the compound account 3.

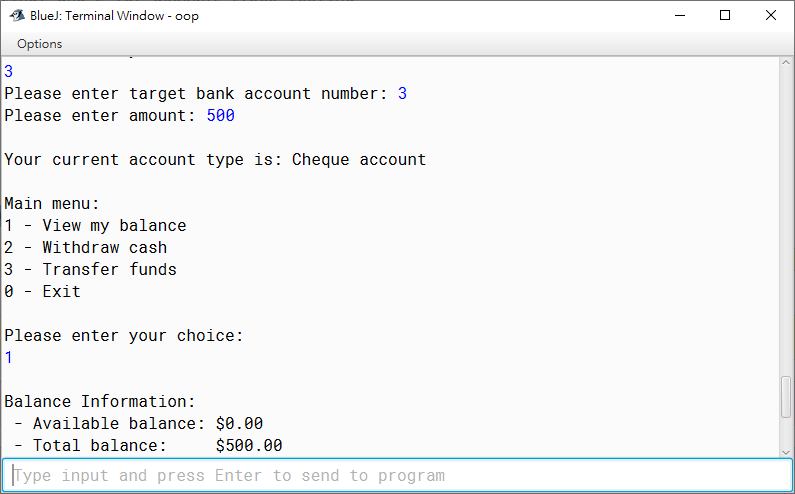


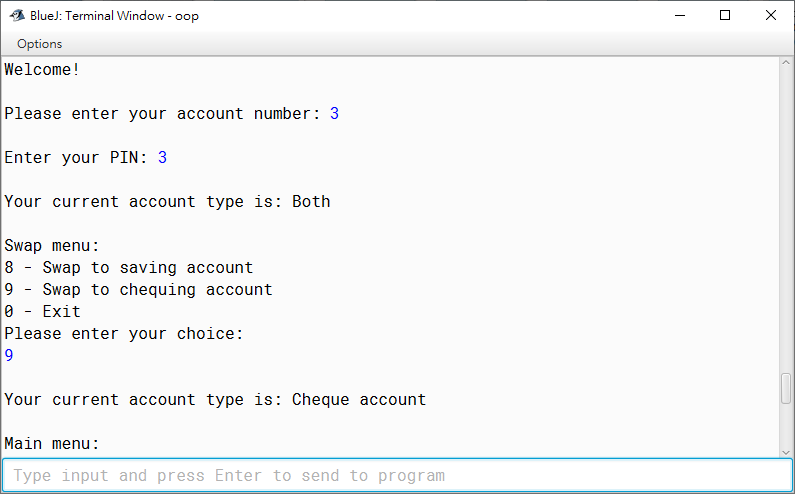
This is the cheque account of the compound account 3.

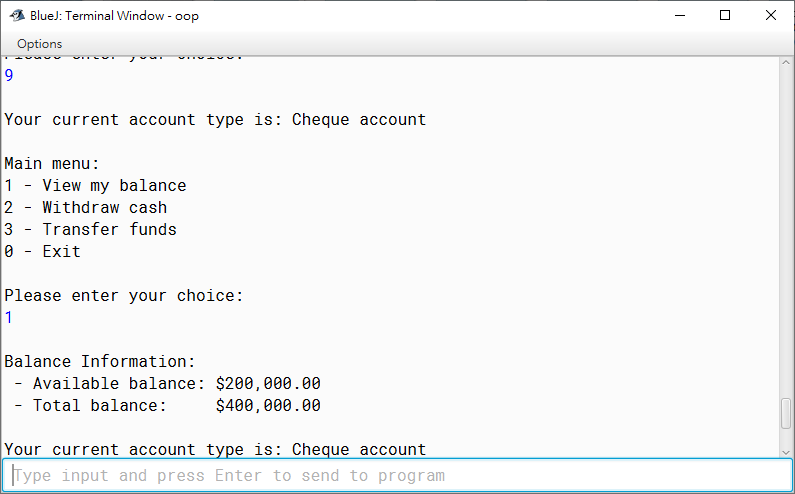


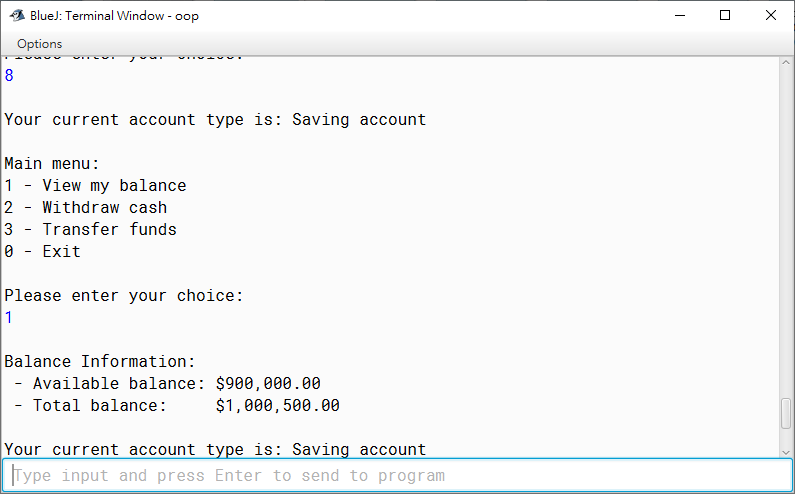
In saving and cheque account page, input 0 can let you back to the swap menu.











Above 5 screenshots show case that transferring $500 to the compound account 3, the amount will automatically save at the saving account of account 3 rather than cheque account.

# Appendix

* The division of job duties

|  |  |
| --- | --- |
| Tsoi Yiu Chik\*\*\* | Coding |
| Lai Jiahao | UML |
| Tong Tsz Huen | Report |
| Cheung Man Hei | Report |
| Fung Hiu Yeung | UML |
| Leung Kwan Yin | Coding |

* Timeline of the work done

|  |  |
| --- | --- |
| Initial class diagram | 21/10 |
| Part (A), (B), (C), (D) | 24/10 |
| Further modification on code | 24/10 |
| Modified class diagram | 28/10 |
| Report | 31/10 |

* Group learning experience

At the coding part, since there is no real-time multi-users coding software in Java language, our groupmates cannot edit and modify the program at the same time. Therefore, we decided to use the cloud drive GitHub to save all our program data. Groupmate who finishes the coding part will upload the initial file to the drive, hence other groupmates download the source file and help with the debugging. We test for the errors by checking the program with valid and invalid data sets. The source code file will keep updating until there is no error, and all the functions are well completed.

To make the work more efficient, we used social software Discord to communicate with each other. This software can do screen sharing, such that groupmate can introduce and explain the frame of their modified program using his screen streaming. It is much easier for us to understand what he had modified in the code because we can raise questions and have a real-time discussion on the misunderstanding parts. Having instant feedback enables us to correct and adjust our program efficiently, hence making the tasks be completed more smoothly.

At the report part, Microsoft word is the best choice for writing a report since we all can edit it at the same time. We can have voice chats in Discord while discussing what to include in the report and do editing on the word file at the same time, having a clearer division of job duties and better communication between groupmates. Furthermore, we can check and comment on other parts and have faster adjustments and corrections.