import java.math.BigDecimal;  
import java.util.Date;  
import java.util.Scanner;  
  
public class Main {  
 public static void main(String[] args) {  
 AccountManager accountManager = new AccountManager();  
 String exit = "e";  
 String deposit = "1";  
 String withdraw = "2";  
 String displayDeposits = "3";  
 String displayWithdrawals = "4";  
 String checking = "1";  
 String saving = "2";  
  
 Customer John = new Customer("John", 1, 500.00, 400, accountManager);  
  
  
 while (true) {  
 System.*out*.println("""  
 Enter the corresponding letter or number to interact with the ATM.  
 1. Deposit  
 2. Withdraw  
 3. Display Deposits  
 4. Display Withdrawals  
 e. Exit""");  
  
 Scanner userInput = new Scanner(System.*in*);  
 String nextLine = userInput.nextLine();  
  
 if (nextLine.equals(deposit)) {  
 depositloop:  
 while (true) {  
 System.*out*.println("""  
 Which account would you like to deposit in?  
 1. Checking  
 2. Saving  
 e. Exit""");  
  
 nextLine = userInput.nextLine();  
  
 if (nextLine.equals(checking) || nextLine.equals(saving)) {  
 while (true) {  
 System.*out*.println("How much would you like to deposit?");  
  
 if (userInput.hasNextDouble()) {  
 double nextDouble = userInput.nextDouble();  
  
 if (nextDouble <= 0) {  
 System.*out*.println("Sorry, please enter an amount to deposit that's greater than " +  
 "$0 next time.");  
 userInput.nextLine();  
 break;  
 }  
 else if (BigDecimal.*valueOf*(nextDouble).scale() > 2) {  
 System.*out*.println("Sorry, please enter an amount to deposit that's 2 decimal " +  
 "spaces or less next time.");  
 userInput.nextLine();  
 break;  
 }  
 else if (nextLine.equals(checking)) {  
 John.deposit(nextDouble, new Date(), Customer.*CHECKING*);  
 John.displayDeposits();  
 break depositloop;  
 }  
 else {  
 John.deposit(nextDouble, new Date(), Customer.*SAVING*);  
 John.displayDeposits();  
 break depositloop;  
 }  
 }  
  
 else {  
 System.*out*.println("Sorry, please enter a sufficient amount next time.");  
 userInput.nextLine();  
 break;  
 }  
 }  
 }  
  
 else if (nextLine.equals(exit)) {  
 System.*out*.println("Returning to the main menu...");  
 break;  
 }  
  
 else {  
 System.*out*.println("Sorry, please enter a valid letter or number.");  
 }  
 }  
 }  
  
 else if (nextLine.equals(withdraw)) {  
 withdrawloop:  
 while (true) {  
 System.*out*.println("""  
 Which account would you like to withdraw from?  
 1. Checking  
 2. Saving  
 e. Exit""");  
  
 nextLine = userInput.nextLine();  
  
 if (nextLine.equals(checking) || nextLine.equals(saving)) {  
 while (true) {  
 System.*out*.println("How much would you like to withdraw?");  
  
 if (userInput.hasNextDouble()) {  
 double nextDouble = userInput.nextDouble();  
  
 if (nextDouble <= 0) {  
 System.*out*.println("Sorry, please enter an amount to withdraw that's greater " +  
 "than $0 next time.");  
 userInput.nextLine();  
 break;  
 }  
 else if (BigDecimal.*valueOf*(nextDouble).scale() > 2) {  
 System.*out*.println("Sorry, please enter an amount to withdraw that's 2 decimal " +  
 "spaces or less next time.");  
 userInput.nextLine();  
 break;  
 }  
 else if (nextLine.equals(checking)) {  
 John.withdraw(nextDouble, new Date(), Customer.*CHECKING*);  
 John.displayWithdraws();  
 break withdrawloop;  
 }  
 else {  
 John.withdraw(nextDouble, new Date(), Customer.*SAVING*);  
 John.displayWithdraws();  
 break withdrawloop;  
 }  
 }  
  
 else {  
 System.*out*.println("Sorry, please enter a sufficient amount next time.");  
 userInput.nextLine();  
 break;  
 }  
 }  
 }  
  
 else if (nextLine.equals(exit)) {  
 System.*out*.println("Returning to the main menu...");  
 break;  
 }  
  
 else {  
 System.*out*.println("Sorry, please enter a valid letter or number.");  
 }  
 }  
 }  
  
 else if (nextLine.equals(displayDeposits)) {  
 System.*out*.println("Displaying deposits...");  
 John.displayDeposits();  
 }  
  
 else if (nextLine.equals(displayWithdrawals)) {  
 System.*out*.println("Displaying withdrawals...");  
 John.displayWithdraws();  
 }  
  
 else if (nextLine.equals(exit)) {  
 System.*out*.println("Thank you for using the ATM.");  
 break;  
 }  
  
 else {  
 System.*out*.println("Sorry, please enter a valid number or letter.");  
 }  
 }  
 }  
}

import java.math.BigDecimal;  
import java.text.DecimalFormat;  
import java.util.ArrayList;  
import java.util.Date;  
  
public class Customer {  
  
 private final ArrayList<Deposit> deposits = new ArrayList<>();  
 private final ArrayList<Withdraw> withdraws = new ArrayList<>();  
 private final String name;  
 private double checkingBalance;  
 private double savingBalance;  
 //private double savingRate;  
 public static final String *CHECKING* = "Checking";  
 public static final String *SAVING* = "Saving";  
 private int overdraftCounter = 0;  
 //private final int overdraft = -100;  
 private static final DecimalFormat *rounder* = new DecimalFormat("0.00");  
 private boolean invalidAccountNumber = false;  
 private boolean invalidCheckingBalance = false;  
 private boolean invalidSavingBalance = false;  
  
  
 // Requires: ArrayList<Integer> array, int value  
 // Modifies: nothing  
 // Effects: Returns true if a value is found in an array  
 public boolean intArrayListContains(ArrayList<Integer> arrayList, int value) {  
  
 for (int currentArrayValue : arrayList) {  
  
 if (currentArrayValue == value) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 // Requires: String name, unique int accountNumber > 0, double checkingBalance and savingBalance up to 2 decimal  
 // places with optional trailing zeroes  
 // Modifies: this  
 // Effects: Constructs a Customer with a name; unique account number greater than 0; checking and saving balances  
 // that have 2 decimal places or less with optional trailing zeroes; and adds the account number to  
 // ArrayList<Integer> accountNumbers. If the account number is less than 1, used by another account, or if the  
 // checkingBalance or savingBalance contains a non-zero number beyond the hundredths place, the Customer's  
 // future deposit or withdrawal will be declined and a reason will be sent to the console  
 Customer(String name, int accountNumber, double checkingBalance, double savingBalance, AccountManager accountManager) {  
  
 this.name = name;  
  
 if (accountNumber <= 0) {  
 System.*out*.println("Sorry " + name + ", please enter an account number that's greater than 0.");  
 this.invalidAccountNumber = true;  
 }  
  
 else if (intArrayListContains(accountManager.getAccountNumbers(), accountNumber)) {  
 System.*out*.println("Sorry " + name + ", please enter a different account number as the account number " +  
 "you've chosen is already being used.");  
 this.invalidAccountNumber = true;  
 }  
  
 else if (BigDecimal.*valueOf*(checkingBalance).scale() > 2) {  
 System.*out*.println("Sorry " + name + ", please enter a checking balance that's 2 decimal spaces or less.");  
 this.invalidCheckingBalance = true;  
 }  
  
 else if (BigDecimal.*valueOf*(savingBalance).scale() > 2) {  
 System.*out*.println("Sorry " + name + ", please enter a saving balance that's 2 decimal spaces or less.");  
 this.invalidSavingBalance = true;  
 }  
  
 else {  
 this.checkingBalance = checkingBalance;  
 this.savingBalance = savingBalance;  
  
 accountManager.addAccountNumber(accountNumber);  
 }  
 }  
  
 // Requires: double amount > 0 with up to 2 decimal places with optional trailing zeroes, Date date, String account  
 // = CHECKING || SAVING  
 // Modifies: this, deposits  
 // Effects: Deposits an amount greater than 0 with up to 2 decimal places with optional trailing zeroes into a  
 // Customer's CHECKING or SAVING account and adds it and a date to ArrayList<Deposit> deposits. If the Customer's  
 // account is invalid from its Construction, the deposit will not go through and a message will be sent to the  
 // console with the relevant reason  
 public void deposit(double amount, Date date, String account) {  
  
 if (this.invalidAccountNumber) {  
 System.*out*.println("Sorry " + this.name + ", the amount could not be deposited because the account " +  
 "number of this account is invalid.");  
 }  
  
 else if (this.invalidCheckingBalance) {  
 System.*out*.println("Sorry " + this.name + ", the amount could not be deposited because the checking " +  
 "balance of this account is invalid.");  
 }  
  
 else if (this.invalidSavingBalance) {  
 System.*out*.println("Sorry " + this.name + ", the amount could not be deposited because the saving " +  
 "balance of this account is invalid.");  
 }  
  
 else if (amount <= 0) {  
 System.*out*.println("Sorry " + this.name + ", please enter an amount to deposit that's greater than $0.");  
 }  
  
 else if (BigDecimal.*valueOf*(amount).scale() > 2) {  
 System.*out*.println("Sorry " + this.name + ", please enter an amount to deposit that's 2 decimal spaces " +  
 "or less.");  
 }  
  
 else {  
  
 if (account.equals(*CHECKING*)) {  
 this.checkingBalance = Math.*round*((this.checkingBalance + amount) \* 100.00) / 100.00;  
 deposits.add(new Deposit(amount, date, *CHECKING*, this.checkingBalance));  
 }  
  
 else if (account.equals(*SAVING*)) {  
 this.savingBalance = Math.*round*((this.savingBalance + amount) \* 100.00) / 100.00;  
 deposits.add(new Deposit(amount, date, *SAVING*, this.savingBalance));  
 }  
  
 else {  
 System.*out*.println("Sorry " + this.name + ", please enter a valid account.");  
 }  
 }  
  
 }  
  
 // Requires: double amount > 0 with up to 2 decimal places with optional trailing zeroes, Date date, String account  
 // = CHECKING || SAVING  
 // Modifies: this, withdraws  
 // Effects: Withdraws an amount greater than 0 with up to 2 decimal places with optional trailing zeroes from a  
 // Customer's CHECKING or SAVING account and adds it and a date to ArrayList<Withdraw> withdraws. If the Customer's  
 // account is invalid from its Construction, the withdrawal will not go through and a message will be sent to the  
 // console with the relevant reason. If the amount withdrawn is greater than the account's balance, a message is  
 // sent to the console and the withdrawal will go through.  
 public void withdraw(double amount, Date date, String account) {  
  
 if (this.invalidAccountNumber) {  
 System.*out*.println("Sorry " + this.name + ", the amount could not be withdrawn because the account " +  
 "number of this account is invalid.");  
 }  
  
 else if (this.invalidCheckingBalance) {  
 System.*out*.println("Sorry " + this.name + ", the amount could not be withdrawn because the checking " +  
 "balance of this account is invalid.");  
 }  
  
 else if (this.invalidSavingBalance) {  
 System.*out*.println("Sorry " + this.name + ", the amount could not be withdrawn because the saving " +  
 "balance of this account is invalid.");  
 }  
  
 else if (amount <= 0) {  
 System.*out*.println("Sorry " + this.name + ", please enter an amount to withdraw that's greater than $0.");  
 }  
  
 else if (BigDecimal.*valueOf*(amount).scale() > 2) {  
 System.*out*.println("Sorry " + this.name + ", please enter an amount to withdraw that's 2 decimal spaces " +  
 "or less.");  
 }  
  
 else {  
  
 while (true) {  
  
 double balance;  
 if (account.equals(*CHECKING*)) {  
 balance = this.checkingBalance;  
 }  
  
 else if (account.equals(*SAVING*)) {  
 balance = this.savingBalance;  
 }  
  
 else {  
 System.*out*.println("Sorry " + this.name + ", please enter a valid account.");  
 break;  
 }  
  
 balance = Math.*round*((balance - amount) \* 100.00) / 100.00;  
 withdraws.add(new Withdraw(amount, date, account, balance));  
  
 if (checkOverdraft(balance)) {  
 System.*out*.println(this.name + ", your withdrawal exceeds your current balance for your " + account  
 + " account. While the money has been withdrawn, you are -$" +  
 *rounder*.format(balance \* -1) + " in the red.");  
 this.overdraftCounter++;  
 }  
  
 if (account.equals(*CHECKING*)) {  
 this.checkingBalance = balance;  
 }  
  
 else {  
 this.savingBalance = balance;  
 }  
  
 break;  
 }  
 }  
 }  
  
 public ArrayList<Deposit> getDeposits() {  
 return deposits;  
 }  
  
 public ArrayList<Withdraw> getWithdraws() {  
 return withdraws;  
 }  
  
 public int getOverdraftCounter() {  
 return overdraftCounter;  
 }  
  
 // Requires: double amount > 0 with up to 2 decimal places with optional trailing zeroes, Date date, String account  
 // = CHECKING || SAVING  
 // Modifies: this  
 // Effects: Checks if a Customer's balance is less than 0 after a withdrawal  
 private boolean checkOverdraft(double amt){  
 return amt < 0;  
 }  
  
 //do not modify  
 public void displayDeposits(){  
 for(Deposit d : deposits){  
 System.*out*.println(d);  
 }  
 }  
 //do not modify  
 public void displayWithdraws(){  
 for(Withdraw w : withdraws){  
 System.*out*.println(w);  
 }  
 }  
}

import java.util.ArrayList;  
  
public class AccountManager {  
 private final ArrayList<Integer> accountNumbers = new ArrayList<>();  
  
 // Requires: int accountNumber > 0  
 // Modifies: this  
 // Effects: Adds an account number to the accountNumbers ArrayList  
 public void addAccountNumber(int accountNumber) {  
 accountNumbers.add(accountNumber);  
 }  
  
 public ArrayList<Integer> getAccountNumbers() {  
 return accountNumbers;  
 }  
}

import org.junit.Test;  
  
import java.util.ArrayList;  
import java.util.Date;  
  
import static org.junit.Assert.*assertEquals*;  
import static org.junit.Assert.*assertTrue*;  
  
public class CustomerTests {  
  
 private final double testAmount = 1;  
 private final Date date = new Date();  
 private Deposit testDeposit;  
 private Withdraw testWithdraw;  
 private final AccountManager accountManager = new AccountManager();  
  
 @Test  
 public void testValidCustomer() {  
 // Create a test Customer with a valid name, account number, and balances  
 Customer testCustomer1 = new Customer("Test1", 1, 500, 400, accountManager);  
  
 // Get deposits and withdraws  
 ArrayList<Deposit> deposits1 = testCustomer1.getDeposits();  
 ArrayList<Withdraw> withdraws1 = testCustomer1.getWithdraws();  
  
 // Assert that there are 0 deposits or withdraws  
 *assertEquals*(0, deposits1.size());  
 *assertEquals*(0, withdraws1.size());  
  
 // Deposit and withdraw money with regard to CHECKING and SAVING  
 testCustomer1.deposit(testAmount, date, Customer.*CHECKING*);  
 testCustomer1.deposit(testAmount, date, Customer.*SAVING*);  
 testCustomer1.withdraw(testAmount, date, Customer.*CHECKING*);  
 testCustomer1.withdraw(testAmount, date, Customer.*SAVING*);  
  
 // Assert that new Deposits and Withdraws have been created  
 *assertEquals*(2, deposits1.size());  
 *assertEquals*(2, withdraws1.size());  
  
 // Assert that testAmount is equal to Deposit.amounts and Withdraw.amounts  
 testDeposit = deposits1.get(0);  
 *assertEquals*(testAmount, testDeposit.getAmount(), 0);  
 testDeposit = deposits1.get(1);  
 *assertEquals*(testAmount, testDeposit.getAmount(), 1);  
 testWithdraw = withdraws1.get(0);  
 *assertEquals*(testAmount, testWithdraw.getAmount(), 0);  
 testWithdraw = withdraws1.get(1);  
 *assertEquals*(testAmount, testWithdraw.getAmount(), 1);  
  
 // Display deposits and withdraws  
 testCustomer1.displayDeposits();  
 testCustomer1.displayWithdraws();  
 }  
  
 @Test  
 public void testInvalidCustomerForInvalidAccountNumberThatIs0OrLower() {  
 // Create test Customers with valid names, invalid account numbers that are 0 or lower, and valid balances  
 Customer testCustomer1 = new Customer("Test1", 0, 500, 400, accountManager);  
 Customer testCustomer2 = new Customer("Test2", -1, 500, 400, accountManager);  
  
 // Get deposits and withdraws  
 ArrayList<Deposit> deposits1 = testCustomer1.getDeposits();  
 ArrayList<Withdraw> withdraws1 = testCustomer1.getWithdraws();  
 ArrayList<Deposit> deposits2 = testCustomer2.getDeposits();  
 ArrayList<Withdraw> withdraws2 = testCustomer2.getWithdraws();  
  
 // Assert that there are 0 deposits or withdraws  
 *assertEquals*(0, deposits1.size());  
 *assertEquals*(0, withdraws1.size());  
 *assertEquals*(0, deposits2.size());  
 *assertEquals*(0, withdraws2.size());  
  
 // Try to deposit and withdraw money with regard to CHECKING and SAVING  
 testCustomer1.deposit(testAmount, date, Customer.*CHECKING*);  
 testCustomer1.deposit(testAmount, date, Customer.*SAVING*);  
 testCustomer1.withdraw(testAmount, date, Customer.*CHECKING*);  
 testCustomer1.withdraw(testAmount, date, Customer.*SAVING*);  
 testCustomer2.deposit(testAmount, date, Customer.*CHECKING*);  
 testCustomer2.deposit(testAmount, date, Customer.*SAVING*);  
 testCustomer2.withdraw(testAmount, date, Customer.*CHECKING*);  
 testCustomer2.withdraw(testAmount, date, Customer.*SAVING*);  
  
 // Assert that deposits and withdraws are empty  
 *assertTrue*(deposits1.isEmpty());  
 *assertTrue*(withdraws1.isEmpty());  
 *assertTrue*(deposits2.isEmpty());  
 *assertTrue*(withdraws2.isEmpty());  
  
 // Try to display deposits and withdraws  
 testCustomer1.displayDeposits();  
 testCustomer1.displayWithdraws();  
 testCustomer2.displayDeposits();  
 testCustomer2.displayWithdraws();  
 }  
  
 @Test  
 public void testInvalidCustomerForInvalidRepeatAccountNumber() {  
 // Create test Customers with valid names, repeating account numbers, and valid balances  
 Customer testCustomer1 = new Customer("Test1", 1, 500, 400, accountManager);  
 Customer testCustomer2 = new Customer("Test2", 1, 500, 400, accountManager);  
  
 // Get deposits and withdraws  
 ArrayList<Deposit> deposits1 = testCustomer1.getDeposits();  
 ArrayList<Withdraw> withdraws1 = testCustomer1.getWithdraws();  
 ArrayList<Deposit> deposits2 = testCustomer2.getDeposits();  
 ArrayList<Withdraw> withdraws2 = testCustomer2.getWithdraws();  
  
 // Assert that there are 0 deposits or withdraws  
 *assertEquals*(0, deposits1.size());  
 *assertEquals*(0, withdraws1.size());  
 *assertEquals*(0, deposits2.size());  
 *assertEquals*(0, withdraws2.size());  
  
 // Try to deposit and withdraw money with regard to CHECKING and SAVING  
 testCustomer1.deposit(testAmount, date, Customer.*CHECKING*);  
 testCustomer1.deposit(testAmount, date, Customer.*SAVING*);  
 testCustomer1.withdraw(testAmount, date, Customer.*CHECKING*);  
 testCustomer1.withdraw(testAmount, date, Customer.*SAVING*);  
 testCustomer2.deposit(testAmount, date, Customer.*CHECKING*);  
 testCustomer2.deposit(testAmount, date, Customer.*SAVING*);  
 testCustomer2.withdraw(testAmount, date, Customer.*CHECKING*);  
 testCustomer2.withdraw(testAmount, date, Customer.*SAVING*);  
  
 // Assert that deposits and withdraws have been made or are empty  
 *assertEquals*(2, deposits1.size());  
 *assertEquals*(2, withdraws1.size());  
 *assertTrue*(deposits2.isEmpty());  
 *assertTrue*(withdraws2.isEmpty());  
  
 // Try to display deposits and withdraws  
 testCustomer1.displayDeposits();  
 testCustomer1.displayWithdraws();  
 testCustomer2.displayDeposits();  
 testCustomer2.displayWithdraws();  
 }  
  
 @Test  
 public void testValidCustomerForValidBalanceThatIs0OrLower() {  
 // Create a test Customer with a valid name, account number, and balances that are 0 or lower  
 Customer testCustomer1 = new Customer("Test1", 1, 0, -1, accountManager);  
  
 // Get deposits and withdraws  
 ArrayList<Deposit> deposits1 = testCustomer1.getDeposits();  
 ArrayList<Withdraw> withdraws1 = testCustomer1.getWithdraws();  
  
 // Assert that there are 0 deposits or withdraws  
 *assertEquals*(0, deposits1.size());  
 *assertEquals*(0, withdraws1.size());  
  
 // Deposit and withdraw money with regard to CHECKING and SAVING  
 testCustomer1.deposit(testAmount, date, Customer.*CHECKING*);  
 testCustomer1.deposit(testAmount, date, Customer.*SAVING*);  
 testCustomer1.withdraw(testAmount, date, Customer.*CHECKING*);  
 testCustomer1.withdraw(testAmount, date, Customer.*SAVING*);  
  
 // Assert that new Deposits and Withdraws have been created  
 *assertEquals*(2, deposits1.size());  
 *assertEquals*(2, withdraws1.size());  
  
 // Assert that testAmount is equal to Deposit.amounts and Withdraw.amounts  
 testDeposit = deposits1.get(0);  
 *assertEquals*(testAmount, testDeposit.getAmount(), 0);  
 testDeposit = deposits1.get(1);  
 *assertEquals*(testAmount, testDeposit.getAmount(), 1);  
 testWithdraw = withdraws1.get(0);  
 *assertEquals*(testAmount, testWithdraw.getAmount(), 0);  
 testWithdraw = withdraws1.get(1);  
 *assertEquals*(testAmount, testWithdraw.getAmount(), 1);  
  
 // Display deposits and withdraws  
 testCustomer1.displayDeposits();  
 testCustomer1.displayWithdraws();  
 }  
  
 @Test  
 public void testValidCustomerForValidBalanceWithMoreThan2DecimalPlaces() {  
 // Create a test Customer with a valid name, account number, and balances with more than 2 decimal places  
 Customer testCustomer1 = new Customer("Test1", 1, 1.000, 0.00000000, accountManager);  
  
 // Get deposits and withdraws  
 ArrayList<Deposit> deposits1 = testCustomer1.getDeposits();  
 ArrayList<Withdraw> withdraws1 = testCustomer1.getWithdraws();  
  
 // Assert that there are 0 deposits or withdraws  
 *assertEquals*(0, deposits1.size());  
 *assertEquals*(0, withdraws1.size());  
  
 // Deposit and withdraw money with regard to CHECKING and SAVING  
 testCustomer1.deposit(testAmount, date, Customer.*CHECKING*);  
 testCustomer1.deposit(testAmount, date, Customer.*SAVING*);  
 testCustomer1.withdraw(testAmount, date, Customer.*CHECKING*);  
 testCustomer1.withdraw(testAmount, date, Customer.*SAVING*);  
  
 // Assert that new Deposits and Withdraws have been created  
 *assertEquals*(2, deposits1.size());  
 *assertEquals*(2, withdraws1.size());  
  
 // Assert that testAmount is equal to Deposit.amounts and Withdraw.amounts  
 testDeposit = deposits1.get(0);  
 *assertEquals*(testAmount, testDeposit.getAmount(), 0);  
 testDeposit = deposits1.get(1);  
 *assertEquals*(testAmount, testDeposit.getAmount(), 1);  
 testWithdraw = withdraws1.get(0);  
 *assertEquals*(testAmount, testWithdraw.getAmount(), 0);  
 testWithdraw = withdraws1.get(1);  
 *assertEquals*(testAmount, testWithdraw.getAmount(), 1);  
  
 // Display deposits and withdraws  
 testCustomer1.displayDeposits();  
 testCustomer1.displayWithdraws();  
 }  
  
 @Test  
 public void testInvalidCustomerForInvalidBalanceWithMoreThan2DecimalPlaces() {  
 // Create test Customers with valid names, account numbers, and invalid balances with more than 2 decimal places  
 Customer testCustomer1 = new Customer("Test1", 1, 1.001, 135.43576, accountManager);  
 Customer testCustomer2 = new Customer("Test2", 2, 1.000, 135.43576, accountManager);  
  
 // Get deposits and withdraws  
 ArrayList<Deposit> deposits1 = testCustomer1.getDeposits();  
 ArrayList<Withdraw> withdraws1 = testCustomer1.getWithdraws();  
 ArrayList<Deposit> deposits2 = testCustomer2.getDeposits();  
 ArrayList<Withdraw> withdraws2 = testCustomer2.getWithdraws();  
  
 // Assert that there are 0 deposits or withdraws  
 *assertEquals*(0, deposits1.size());  
 *assertEquals*(0, withdraws1.size());  
 *assertEquals*(0, deposits2.size());  
 *assertEquals*(0, withdraws2.size());  
  
 // Try to deposit and withdraw money with regard to CHECKING and SAVING  
 testCustomer1.deposit(testAmount, date, Customer.*CHECKING*);  
 testCustomer1.deposit(testAmount, date, Customer.*SAVING*);  
 testCustomer1.withdraw(testAmount, date, Customer.*CHECKING*);  
 testCustomer1.withdraw(testAmount, date, Customer.*SAVING*);  
 testCustomer2.deposit(testAmount, date, Customer.*CHECKING*);  
 testCustomer2.deposit(testAmount, date, Customer.*SAVING*);  
 testCustomer2.withdraw(testAmount, date, Customer.*CHECKING*);  
 testCustomer2.withdraw(testAmount, date, Customer.*SAVING*);  
  
 // Assert that deposits and withdraws are empty  
 *assertTrue*(deposits1.isEmpty());  
 *assertTrue*(withdraws1.isEmpty());  
 *assertTrue*(deposits2.isEmpty());  
 *assertTrue*(withdraws2.isEmpty());  
  
 // Try to display deposits and withdraws  
 testCustomer1.displayDeposits();  
 testCustomer1.displayWithdraws();  
 testCustomer2.displayDeposits();  
 testCustomer2.displayWithdraws();  
 }  
}

import org.junit.Before;  
import org.junit.Test;  
  
import java.util.ArrayList;  
import java.util.Date;  
  
import static org.junit.Assert.*assertEquals*;  
  
public class DepositWithdrawTests {  
  
 private final Date date = new Date();  
 private Withdraw testWithdraw;  
 private Withdraw finalCheckingBalance;  
 private Withdraw finalSavingBalance;  
 private double totalTestAmount = 0;  
 private Customer testCustomer1;  
 private final double checkingBalance = 500;  
 private final double savingBalance = 400;  
 private final AccountManager accountManager = new AccountManager();  
  
 @Before  
 public void setUp() {  
 // Create a test Customer with a valid name, account number, and balances  
 testCustomer1 = new Customer("Test1", 1, checkingBalance, savingBalance, accountManager);  
 }  
  
 @Test  
 public void testValidAmount() {  
 // Get deposits and withdraws  
 ArrayList<Deposit> deposits1 = testCustomer1.getDeposits();  
 ArrayList<Withdraw> withdraws1 = testCustomer1.getWithdraws();  
  
 // Assert that there are 0 deposits or withdraws  
 *assertEquals*(0, deposits1.size());  
 *assertEquals*(0, withdraws1.size());  
  
 // Create an array of valid test amounts  
 double[] testAmounts = {1, 1.1, 1.10, 1.01, 1.00, 1.000, 1.110, 1.50, .01, 00001.00, 00.01, 9999999.99};  
  
 // Deposit money in CHECKING  
 for (double testAmount : testAmounts) {  
 testCustomer1.deposit(testAmount, date, Customer.*CHECKING*);  
 }  
  
 // Withdraw money from CHECKING  
 for (double testAmount : testAmounts) {  
 testCustomer1.withdraw(testAmount, date, Customer.*CHECKING*);  
 }  
  
 // Assert that new Deposits and Withdraws have been created  
 *assertEquals*(testAmounts.length, deposits1.size());  
 *assertEquals*(testAmounts.length, withdraws1.size());  
  
 // Assert that testAmount is equal to Deposit.amounts and Withdraw.amounts  
 Deposit testDeposit;  
 for (int i = 0; i < testAmounts.length; i++) {  
 testDeposit = deposits1.get(i);  
 *assertEquals*(testAmounts[i], testDeposit.getAmount(), 0);  
 testWithdraw = withdraws1.get(i);  
 *assertEquals*(testAmounts[i], testWithdraw.getAmount(), 0);  
 }  
  
 // Deposit money in SAVING  
 for (double testAmount : testAmounts) {  
 testCustomer1.deposit(testAmount, date, Customer.*SAVING*);  
 }  
  
 // Withdraw money from SAVING  
 for (double testAmount : testAmounts) {  
 testCustomer1.withdraw(testAmount, date, Customer.*SAVING*);  
 }  
  
 // Assert that double the amount of Deposits and Withdraws have been created  
 *assertEquals*(testAmounts.length \* 2, deposits1.size());  
 *assertEquals*(testAmounts.length \* 2, withdraws1.size());  
  
 // Assert that testAmount is equal to Deposit.amounts and Withdraw.amounts for SAVING  
 for (int i = 0; i < testAmounts.length; i++) {  
 testDeposit = deposits1.get(i + testAmounts.length);  
 *assertEquals*(testAmounts[i], testDeposit.getAmount(), 0);  
 testWithdraw = withdraws1.get(i + testAmounts.length);  
 *assertEquals*(testAmounts[i], testWithdraw.getAmount(), 0);  
 }  
  
 // Assert that current balance in CHECKING and SAVING are equivalent to initial balance  
 finalCheckingBalance = withdraws1.get(withdraws1.size() / 2 - 1);  
 *assertEquals*(checkingBalance, finalCheckingBalance.getBalance(), 0);  
 finalSavingBalance = withdraws1.get(withdraws1.size() - 1);  
 *assertEquals*(savingBalance, finalSavingBalance.getBalance(), 0);  
  
 // Display deposits and withdraws  
 testCustomer1.displayDeposits();  
 testCustomer1.displayWithdraws();  
 }  
  
 @Test  
 public void testInvalidAmount() {  
 // Get deposits and withdraws  
 ArrayList<Deposit> deposits1 = testCustomer1.getDeposits();  
 ArrayList<Withdraw> withdraws1 = testCustomer1.getWithdraws();  
  
 // Assert that there are 0 deposits or withdraws  
 *assertEquals*(0, deposits1.size());  
 *assertEquals*(0, withdraws1.size());  
  
 // Create an array of invalid test amounts  
 double[] testAmounts = {0, -1, 0.001, .00, 0.00, 0.0, 0.000, 1.1101, 1.124, -.0, -.01, 0.00000000001, 100.001};  
  
 // Deposit and withdraw money with regard to CHECKING  
 for (double testAmount : testAmounts) {  
 testCustomer1.deposit(testAmount, date, Customer.*CHECKING*);  
 testCustomer1.withdraw(testAmount, date, Customer.*CHECKING*);  
 }  
  
 // Assert that deposits and withdraws are empty  
 *assertEquals*(0, deposits1.size());  
 *assertEquals*(0, withdraws1.size());  
  
 // Deposit and withdraw money with regard to SAVING  
 for (double testAmount : testAmounts) {  
 testCustomer1.deposit(testAmount, date, Customer.*SAVING*);  
 testCustomer1.withdraw(testAmount, date, Customer.*SAVING*);  
 }  
  
 // Assert that deposits and withdraws are empty  
 *assertEquals*(0, deposits1.size());  
 *assertEquals*(0, withdraws1.size());  
  
 // Try to display deposits and withdraws  
 testCustomer1.displayDeposits();  
 testCustomer1.displayWithdraws();  
 }  
  
 @Test  
 public void testInvalidAccount() {  
 // Create a test Customer with a valid name, an invalid repeat account number, and valid balances  
 Customer testCustomer2 = new Customer("Test2", 1, 500, 400, accountManager);  
  
 // Get deposits and withdraws  
 ArrayList<Deposit> deposits1 = testCustomer2.getDeposits();  
 ArrayList<Withdraw> withdraws1 = testCustomer2.getWithdraws();  
  
 // Assert that there are 0 deposits or withdraws  
 *assertEquals*(0, deposits1.size());  
 *assertEquals*(0, withdraws1.size());  
  
 // Create an array of valid test amounts  
 double[] testAmounts = {1, 1.1, 1.10, 1.01, 1.00, 1.000, 1.110, 1.50, .01, 00001.00, 00.01, 9999999.99};  
  
 // Deposit and withdraw money with regard to an invalid account  
 for (double testAmount : testAmounts) {  
 testCustomer2.deposit(testAmount, date, "a");  
 testCustomer2.withdraw(testAmount, date, "CHECKING");  
 }  
  
 // Assert that deposits and withdraws are empty  
 *assertEquals*(0, deposits1.size());  
 *assertEquals*(0, withdraws1.size());  
  
 // Try to display deposits and withdraws  
 testCustomer2.displayDeposits();  
 testCustomer2.displayWithdraws();  
 }  
  
 @Test  
 public void testOverdraft() {  
 // Get withdraws  
 ArrayList<Withdraw> withdraws1 = testCustomer1.getWithdraws();  
  
 // Assert that there are 0 withdraws  
 *assertEquals*(0, withdraws1.size());  
  
 // Create an array of valid test amounts that will cause overdrafts  
 double[] testAmounts = {checkingBalance + 1, savingBalance + 1, checkingBalance \* 5, savingBalance \* 5};  
  
 // Withdraw money from CHECKING  
 for (double testAmount : testAmounts) {  
 testCustomer1.withdraw(testAmount, date, Customer.*CHECKING*);  
 }  
  
 // Assert that new Withdraws have been created  
 *assertEquals*(testAmounts.length, withdraws1.size());  
  
 // Assert that testAmount is equal to Withdraw.amounts  
 for (int i = 0; i < testAmounts.length; i++) {  
 testWithdraw = withdraws1.get(i);  
 *assertEquals*(testAmounts[i], testWithdraw.getAmount(), 0);  
 }  
  
 // Withdraw money from SAVING  
 for (double testAmount : testAmounts) {  
 testCustomer1.withdraw(testAmount, date, Customer.*SAVING*);  
 }  
  
 // Assert that double the amount of Withdraws have been created  
 *assertEquals*(testAmounts.length \* 2, withdraws1.size());  
  
 // Assert that testAmount is equal to Deposit.amounts and Withdraw.amounts for SAVING  
 for (int i = 0; i < testAmounts.length; i++) {  
 testWithdraw = withdraws1.get(i + testAmounts.length);  
 *assertEquals*(testAmounts[i], testWithdraw.getAmount(), 0);  
 }  
  
 // Assert that current balance in CHECKING and SAVING are equivalent to initial balance - testAmounts  
 for (double testAmount : testAmounts) {  
 totalTestAmount += testAmount;  
 }  
 finalCheckingBalance = withdraws1.get(withdraws1.size() / 2 - 1);  
 *assertEquals*(checkingBalance - totalTestAmount, finalCheckingBalance.getBalance(), 0);  
 finalSavingBalance = withdraws1.get(withdraws1.size() - 1);  
 *assertEquals*(savingBalance - totalTestAmount, finalSavingBalance.getBalance(), 0);  
  
 // Assert that overdraft messages have been sent  
 *assertEquals*(withdraws1.size(), testCustomer1.getOverdraftCounter());  
  
 // Display deposits and withdraws  
 testCustomer1.displayWithdraws();  
 }  
}

import java.util.Date;  
import java.text.DecimalFormat;  
public class Withdraw {  
  
 private final double amount;  
 private final Date date;  
 private final String account;  
 private final double balance;  
  
 private static final DecimalFormat *rounder* = new DecimalFormat("0.00");  
  
 // Requires: double amount > 0 with up to 2 decimal places with optional trailing zeroes, Date date, String account  
 // = CHECKING || SAVING, double balance with up to 2 decimal places with optional trailing zeroes  
 // Modifies: this  
 // Effects: Constructs a Withdraw containing an amount greater than 0 with up to 2 decimal places with optional  
 // trailing zeroes; a Date date; a CHECKING or SAVING account; and a balance with up to 2 decimal places with  
 // optional trailing zeroes  
 Withdraw(double amount, Date date, String account, double balance){  
 this.amount = amount;  
 this.date = date;  
 this.account = account;  
 this.balance = balance;  
 }  
  
 // Requires: double amount > 0 with up to 2 decimal places with optional trailing zeroes, Date date, String account  
 // = CHECKING || SAVING, double balance with up to 2 decimal places with optional trailing zeroes  
 // Modifies: this  
 // Effects: Converts a Withdraw to a String, rounds all decimal places to 2, and sends it to the console. If balance  
 // is less than 0, adds a negative sign in front of the dollar sign  
 public String toString() {  
 String negativeSign = "";  
 String typedBalance;  
 if (balance > 0) {  
 typedBalance = *rounder*.format(balance);  
 }  
 else {  
 typedBalance = *rounder*.format(balance \* -1);  
 negativeSign = "-";  
 }  
 return "Withdrawal of: $" + *rounder*.format(amount) +  
 " Date: " + date +  
 " into account: " + account +  
 " Current Balance in " + account + " is " + negativeSign + "$" + typedBalance;  
 }  
  
 public double getAmount() {  
 return amount;  
 }  
  
 public double getBalance() {  
 return balance;  
 }  
}

import java.util.Date;  
import java.text.DecimalFormat;  
  
public class Deposit {  
 private final double amount;  
 private final Date date;  
 private final String account;  
 private final double balance;  
 private static final DecimalFormat *rounder* = new DecimalFormat("0.00");  
  
 // Requires: double amount > 0 with up to 2 decimal places with optional trailing zeroes, Date date, String account  
 // = CHECKING || SAVING, double balance with up to 2 decimal places with optional trailing zeroes  
 // Modifies: this  
 // Effects: Constructs a Deposit containing an amount greater than 0 with up to 2 decimal places with optional  
 // trailing zeroes; a Date date; a CHECKING or SAVING account; and a balance with up to 2 decimal places with  
 // optional trailing zeroes  
 Deposit(double amount, Date date, String account, double balance){  
 this.amount = amount;  
 this.date = date;  
 this.account = account;  
 this.balance = balance;  
 }  
  
 // Requires: double amount > 0 with up to 2 decimal places with optional trailing zeroes, Date date, String account  
 // = CHECKING || SAVING, double balance with up to 2 decimal places with optional trailing zeroes  
 // Modifies: this  
 // Effects: Converts a Deposit to a String, rounds all decimal places to 2, and sends it to the console. If balance  
 // is less than 0, adds a negative sign in front of the dollar sign  
 public String toString(){  
 String negativeSign = "";  
 String typedBalance;  
 if (balance > 0) {  
 typedBalance = *rounder*.format(balance);  
 }  
 else {  
 typedBalance = *rounder*.format(balance \* -1);  
 negativeSign = "-";  
 }  
 return "Deposit of: $" + *rounder*.format(amount) +  
 " Date: " + date +  
 " into account: " + account +  
 " Current Balance in " + account + " is " + negativeSign + "$" + typedBalance;  
 }  
  
 public double getAmount() {  
 return amount;  
 }  
}