Bank Report

Object Oriented Programming

December 18

Student Number: C20396243
Student Name: Aayush Gaur
Course Code: DT228-2 / TU856-2

Project Classes

➢ Customer

Stores all information about the customer.

Attributes:

- ID *(customer_id)*
- Name *(name)*
- Age *(age)*
- PIN *(__pin) private*
- List of Accounts *(accounts[])*

Methods:

- **__init__(self, customer_id, name, age, pin, acc_id=None)**
  Initializes all the attributes with the parameters passed when creating the object. The ***Accounts list*** is empty by default.

- **get_pin(self)**
  Get method to get the value of the private attribute *(__pin)*

- **set_pin(self)**
  Set method to set the value of the private attribute *(__pin)*

- **pin_verify(self)**
  Checks PIN entered by the customer with their actual PIN and returns True or False.

- **change_pin(self)**
  Changes the customer's PIN using set_pin() after verifying his current PIN using pin_verify().

- **add_acc(self, acc_id)**
  Adds an account to the customer's account list *(accounts[]).*

- **print_balance(self)**
  Prints the Account ID, Balance and the Type of Account of all the accounts the customer currently holds. If he has no accounts display the message “You have no accounts”.

- **choose_account(self)**
  Displays the list of accounts the customer hold and prompts him to choose an account.
  It returns the index of the Account that the customer chooses.
  Has -1 as the return value if the customer decides to cancel choosing instead.

- **close_account(self)**
  Closes a Customer's Account after verifying his PIN using *self.pin_verify().*
  It makes the user choose an account from his accounts list using *choose_account()*. It then deletes the selected account by using the *pop()* function on the accounts list.

- **__str__()**
  Returns a string with the Customer ID, Name, Age and a list of accounts that the customer holds.

➢ Account

Stores all information about an account. Has functions to be used for all types of Accounts.

Attributes:

- **ID** *(acc_id)*
- **Balance** *(balance)*
- **List of transactions** *(transactions[])*

Methods:

- **__init__(self, acc_id, balance=0.0, transactions=None)**
  Initializes all the attributes with the parameters passed when creating the object. ***Balance*** has a default value of 0.0 and the ***Transactions list*** is empty by default.

- **deposit(self, amount)**
  Deposits the amount passed into the account.
  It first checks if the amount passed is a positive value and displays an error message if not.
  It adds the amount passed to the ***balance*** attribute of the account calling the method (***self***).
  It also adds a deposit transaction record to the current account.

- **withdraw(self, amount)**
  Withdraws amount from an account. **Polymorphism** is used as it checks for the type of Account Object passed (***Introspection***).
  If the Account is of the class:
  - **SavingAccount** - It checks if the last withdraw or transfer transaction was made in the last 30 days. If the transaction is made within 30 days it shows an error message and prevents the withdrawal.
    It also prevents the withdrawal if the resultant balance is less than 0.

  - **CurrentAccount** - It prevents the withdrawal if the resultant balance is less than the minimum balance limit. No monthly transaction limit.

  It also adds a withdraw transaction record to the current account.

- **print_transactions(self)**
  Prints the last 5 transactions made by the account. If no transactions have been made it displays “No transactions on this Account”.

- **__add__(self, param)**
  Operator overloading: Adds the param to the balance of the account.

- **__radd__(self, param)**
  Reverses the parameter if __add__ fails. Adds the param to the balance of the account.

- **__sub__(self, param)**
  Operator overloading: Subtract the param from the balance of the account.

- **__rsub__(self, param)**
  Reverses the parameter if __rsub__ fails. Subtract the param from the balance of the account.

- **__eq__(self, param)**
  Compare two account objects for equality based on Account ID, return Boolean.

- **__str__(self)**
  Returns a string with the Account ID and Balance.

➢ SavingAccount

It is a subclass of Account. It inherits all the attributes and functions from Account and has attributes and functions which are used for a Savings Account.

Attributes:

- **ID***
- **Balance***
- **List of transactions***
- **Account Type** (*accType*)

* Inherited from Account Class

Methods:

- **__init__(self, acc_id, balance=0.0, transactions=None)**
  Initializes all the attributes using the ***Account.__init__()*** . Adds a new attribute accType = “Savings”.

- **deposit(self)***
- **withdraw(self)***
- **print_transactions(self)***

- **transfer(self, amount, receiver_acc)**
  Transfers amount from current account *(self)* into another account *(receiver_acc)*.
  It checks when the last “withdraw” or “transfer” was made and calculates the days passed between that transaction and today.
  If the days passed is less than 30, it shows you a message saying “you can only withdraw or transfer once every 30 days”.
  It also checks if the resultant balance is less than 0.
  If the resultant balance is not less than zero, it transfers the amount from the current account to the receiver account passed.
  It also adds a transfer transaction record to both the current account and receiver account.

- **__str(self)__**
  Returns a string with the Account ID, Balance and Account type.
  (Account.__str__ + self.accType)

➢ CheckingAccount

It is a subclass of Account. It inherits all the attributes and functions from Account and has attributes and functions which are used for a Checking Account.

Attributes:

- **ID***
- **Balance***
- **List of transactions***
- **Account Type** (*accType*)
- **Minimum Balance** *(minimum_balance)*

* Inherited from Account Class

Methods:

- **__init__(self, acc_id, balance=0.0, transactions=None, minimum_balance=-1000)**
  Initializes all the attributes using the ***Account.__init__()*** . Adds two new attributes accType = "Checking" and minimum_balance.

- **deposit(self)***
- **withdraw(self)***
- **print_transactions(self)***

- **transfer(self, amount, receiver_acc)**
  Transfers amount from current account *(self)* into another account *(receiver_acc)*.
  It checks if the amount passed is a positive value. If not, it shows an error message and exits the method.
  It checks if the resultant balance is less than the minimum balance limit. If the resultant balance is not less than the minimum balance limit, it transfers the amount from the current account to the receiver account passed. It also adds a transfer transaction record to both the current account and receiver account.

- **__str(self)__**
  Returns a string with the Account ID, Balance and Account type.
  (Account.__str__ + self.accType)

Project Functions

➢ get_next_trx_id()

This returns the next transaction ID. It check the current value of the global variable ***trx_id.*** Increment the value by one (the last 3 digits). Return the value.

➢ create_savings_account()

Checks the age of the customer and creates a savings account for the customer if he is eligible (age >= 14).

➢ create_checking_account()

Checks the age of the customer and creates a checking account for the customer if he is eligible (age >= 18).

➢ update_files(customers)

Updates the files:

customers.txt	-	With the information about each Customer
accounts.txt	-	With the information about each Accounts
accountsTransactions.txt	-	With the information about each Transaction

➢ create_bank_obj(customers, accounts)

Creates previous customer objects and Account objects with information from the files.
Store all the customer objects into the ***customer*** dictionary passed, with the Customer ID as key.
Store all the customer objects into the ***accounts*** dictionary passed, with the Account ID as key.

➢ menu(customer, allcustomers, accounts)

Displays the main menu after logging in and calls the respective functions as the user chooses.
Parameters:

- customer: Customer object of the customer who has logged in.
- allcustomers: customer dictionary containing all the existing customer objects.
- accounts: account dictionary containing all the existing account objects.

➢ main()

Main function.
Calls the ***create_bank_obj()*** to create all the previous customers and accounts.
Displays a menu for the user to:

- **Create an account**
  Creates a new customer object based on information taken from the user.
  Display the customer account information after successful object creation.

- **Login**
  Ask the user to enter his customer ID and verify his PIN.
  Call the ***menu()*** after successful login.

- **Exit**
  Exit the menu loop.

After every loop call the ***update_files()*** to update the files with all the information.

Project Manual

At the start you are displayed 3 option to choose from.

1. Create an account *(Customer Account)*
2. Login
3. Exit.

Choose any option number and follow the instructions displaying on the terminal.
You will need your Customer ID and PIN to login.

After logging in you are shown a main menu:

1. Create a new account *(Bank Account: Savings Account or Checking Account)*
2. View Balance
3. Deposit
4. Withdraw
5. Transfer
6. View Transactions
7. Close an Account
8. Change PIN
9. Logout

Choose any option number and follow the instructions on the terminal.

Warning:
Be sure to Logout after you are finished. The information in the files is updated only after a user logs out.

If a customer decides to close all his bank accounts (leaving the bank) his *Customer Account* should also be deleted. It doesn't make sense to have a customer at the bank without any accounts. The program has taken this case into account and deletes the customer accounts without any bank accounts.

This is done when updating the files after the user logs out.
In the ***update_file()*** it only writes to the file only if the customer has a bank account attached to it.

Due to the above process, after creating a new customer account, the user has to login to his account and create a new bank account. Or else the newly created customer account will be deleted if no bank account has been created.

Project Contribution

I did this project all by myself.

Difficulties

- The format of storing information in the files was tricky. A customer could have one or many accounts.
  To maintain the format of the information in the file I came up with the idea to store it as:
  <Customer ID> <Name> <Age> <PIN> <Account ID>
  With only one account ID per line. This way each line can tell you which account belongs to which customer.

- Storing all the customer objects was a problem. Just a normal array or list of objects would not work as then I have to iterate through the whole list to find a particular customer account.
  This was solved by using a dictionary to store all the objects with their unique ID ***(customer ID)*** as the key.

- Creating customer objects with many accounts.
  This was solved by checking if the customer ID already existed.
  By checking if the ID was already a key in the customer dictionary.
  If it did the new account object was appended to the accounts list of the customer.
  Else a new customer was created with that account object by passing the account object as an argument while creating the customer object.

- Creating a Unique ID for every object was tricky.
  This was solved by checking what the max ID was in the customers/accounts dictionary and incrementing it by one.
  For the transaction ID, the max transaction ID is found while loading in the transactions from the files. Then it is stored in a global variable. Thus every time a new transaction ID is required the function ***get_next_trx_id()*** increments the global variable by 1 and returns that value.

- Restricting the withdraw and transfer to once every 30 days for a Savings Account.
  To get the days between the last withdraw or transfer transaction and today was difficult.
  DateTime was imported to overcome this.
  First I had to check for when the last withdraw or transfer was made. This date was converted into datetime object using ***strptime()***. The difference between the date and today in days was calculate using:
  (date.today() – last_trx_date).days
  The result was compared to 30 and the appropriate message was displayed.

- Updating the files after a customer had removed an account.
  At first I looped through the customers and accounts dictionary and printed it to their respective files. This caused the account and transaction information about the deleted accounts to remain as these objects were not popped from the dictionary (only from the accounts list in customer object).
  This was fixed by looping through the customer objects in customer dictionary only.
  Account information was accessed through customer.accounts[].
  Transactions information was accessed through customer.accounts[].transactions[].
  This way deleting an account or customer deletes all the information related to it from all 3 files.