Banking System

Instructions

- Submitting assignments should be a single file or through git hub link shared with trainer and hexavarsity.
- Each assignment builds upon the previous one, and by the end, you will have a comprehensive Banking System implemented in Java/C#/Python with a strong focus on SQL schema design, control flow statements, loops, arrays, collections, and database interaction.
- Follow object-oriented principles throughout the Java programming assignments. Use classes and objects to model real-world entities, encapsulate data and behaviour, and ensure code reusability.
- Throw user defined exception from method and handle in the main method**.**
- The following **Directory structure** is to be followed in the application.
    - **entity/model**
        - Create entity classes in this package. All entity class should not have any business logic.
    - **dao**
        - Create Service Provider interface/abstract class to showcase functionalities.
        - Create the implementation class for the above interface/abstract class with db interaction.
    - **exception**
        - Create user defined exceptions in this package and handle exceptions whenever needed.
    - **util**
        - Create a **DBPropertyUtil** class with a static function which takes property file name as parameter and returns connection string.
        - Create a **DBConnUtil** class which holds **static method** which takes connection string as parameter file and returns **connection object**.
    - **main**
        - Create a class MainModule and demonstrate the functionalities in a menu driven application.

You are tasked with creating an advanced banking system that includes various types of accounts, such as savings and current accounts. The system should support account creation, deposits, withdrawals, and interest calculations.

Database Tables

1. **Customers:**
    - **customer_id (Primary Key)**
    - first_name
    - last_name
    - DOB (Date of Birth)
    - email
    - phone_number

- address

2. **Accounts:**
    - **account_id (Primary Key)**
    - **customer_id (Foreign Key)**
    - account_type (e.g., savings, current, zero_balance)
    - balance
3. **Transactions:**
    - **transaction_id (Primary Key)**
    - **account_id (Foreign Key)**
    - transaction_type (e.g., deposit, withdrawal, transfer)
    - amount
    - transaction_date

Tasks 1: Database Design:

1. Create the database named "HMBank"
2. Define the schema for the Customers, Accounts, and Transactions tables based on the provided schema.
4. Create an ERD (Entity Relationship Diagram) for the database.
5. Create appropriate Primary Key and Foreign Key constraints for referential integrity.
6. Write SQL scripts to create the mentioned tables with appropriate data types, constraints, and relationships.
    - Customers
    - Accounts
    - Transactions

Tasks 2: Select, Where, Between, AND, LIKE:

1. Insert at least 10 sample records into each of the following tables.
    - Customers
    - Accounts
    - Transactions
2. Write SQL queries for the following tasks:
    1. Write a SQL query to retrieve the name, account type and email of all customers.
    2. Write a SQL query to list all transaction corresponding customer.
    3. Write a SQL query to increase the balance of a specific account by a certain amount.
    4. Write a SQL query to Combine first and last names of customers as a full_name.
    5. Write a SQL query to remove accounts with a balance of zero where the account type is savings.
    6. Write a SQL query to Find customers living in a specific city.
    7. Write a SQL query to Get the account balance for a specific account.
    8. Write a SQL query to List all current accounts with a balance greater than $1,000.
    9. Write a SQL query to Retrieve all transactions for a specific account.

10. Write a SQL query to Calculate the interest accrued on savings accounts based on a given interest rate.
11. Write a SQL query to Identify accounts where the balance is less than a specified overdraft limit.
12. Write a SQL query to Find customers not living in a specific city.

Tasks 3: Aggregate functions, Having, Order By, GroupBy and Joins:

1. Write a SQL query to Find the average account balance for all customers.
2. Write a SQL query to Retrieve the top 10 highest account balances.
3. Write a SQL query to Calculate Total Deposits for All Customers in specific date.
4. Write a SQL query to Find the Oldest and Newest Customers.
5. Write a SQL query to Retrieve transaction details along with the account type.
6. Write a SQL query to Get a list of customers along with their account details.
7. Write a SQL query to Retrieve transaction details along with customer information for a specific account.
8. Write a SQL query to Identify customers who have more than one account.
9. Write a SQL query to Calculate the difference in transaction amounts between deposits and withdrawals.
10. Write a SQL query to Calculate the average daily balance for each account over a specified period.
11. Calculate the total balance for each account type.
12. Identify accounts with the highest number of transactions order by descending order.
13. List customers with high aggregate account balances, along with their account types.
14. Identify and list duplicate transactions based on transaction amount, date, and account.

Tasks 4: Subquery and its type:

1. Retrieve the customer(s) with the highest account balance.
2. Calculate the average account balance for customers who have more than one account.
3. Retrieve accounts with transactions whose amounts exceed the average transaction amount.
4. Identify customers who have no recorded transactions.
5. Calculate the total balance of accounts with no recorded transactions.
6. Retrieve transactions for accounts with the lowest balance.
7. Identify customers who have accounts of multiple types.
8. Calculate the percentage of each account type out of the total number of accounts.
9. Retrieve all transactions for a customer with a given customer_id.
10. Calculate the total balance for each account type, including a subquery within the SELECT clause.

Banking System

Control Structure

Task 1: Conditional Statements

In a bank, you have been given the task is to create a program that checks if a customer is eligible for a loan based on their credit score and income. The eligibility criteria are as follows:

- Credit Score must be above 700.
- Annual Income must be at least $50,000.

Tasks:

1. Write a program that takes the customer's credit score and annual income as input.
2. Use conditional statements (if-else) to determine if the customer is eligible for a loan.
3. Display an appropriate message based on eligibility.

Task 2: Nested Conditional Statements

Create a program that simulates an ATM transaction. Display options such as "Check Balance," "Withdraw," "Deposit,". Ask the user to enter their current balance and the amount they want to withdraw or deposit. Implement checks to ensure that the withdrawal amount is not greater than the available balance and that the withdrawal amount is in multiples of 100 or 500. Display appropriate messages for success or failure.

Task 3: Loop Structures

You are responsible for calculating compound interest on savings accounts for bank customers. You need to calculate the future balance for each customer's savings account after a certain number of years.

Tasks:

1. Create a program that calculates the future balance of a savings account.
2. Use a loop structure (e.g., for loop) to calculate the balance for multiple customers.
3. Prompt the user to enter the initial balance, annual interest rate, and the number of years.
4. Calculate the future balance using the formula:

   *future_balance = initial_balance * (1 + annual_interest_rate/100)^years.*
5. Display the future balance for each customer.

Task 4: Looping, Array and Data Validation

You are tasked with creating a program that allows bank customers to check their account balances. The program should handle multiple customer accounts, and the customer should be able to enter their account number, balance to check the balance.

Tasks:

1. Create a Python program that simulates a bank with multiple customer accounts.
2. Use a loop (e.g., while loop) to repeatedly ask the user for their account number and balance until they enter a valid account number.
3. Validate the account number entered by the user.
4. If the account number is valid, display the account balance. If not, ask the user to try again.

Task 5: Password Validation

Write a program that prompts the user to create a password for their bank account. Implement if conditions to validate the password according to these rules:

- The password must be at least 8 characters long.
- It must contain at least one uppercase letter.
- It must contain at least one digit.
- Display appropriate messages to indicate whether their password is valid or not.

Task 6: Password Validation

Create a program that maintains a list of bank transactions (deposits and withdrawals) for a customer. Use a while loop to allow the user to keep adding transactions until they choose to exit. Display the transaction history upon exit using looping statements.

OOPS, Collections and Exception Handling

Task 7: Class & Object

1. Create a `Customer` class with the following confidential attributes:
    - Attributes
        - Customer ID
        - First Name
        - Last Name
        - Email Address
        - Phone Number
        - Address
    - Constructor and Methods
        - Implement default constructors and overload the constructor with Customer attributes, generate getter and setter, (print all information of attribute) methods for the attributes.
2. Create an `Account` class with the following confidential attributes:
    - Attributes
        - Account Number
        - Account Type (e.g., Savings, Current)
        - Account Balance
    - Constructor and Methods
        - Implement default constructors and overload the constructor with Account attributes,
        - Generate getter and setter, (print all information of attribute) methods for the attributes.
        - Add methods to the `Account` class to allow deposits and withdrawals.
            - deposit(amount: float): Deposit the specified amount into the account.

- withdraw(amount: float): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.
- calculate_interest(): method for calculating interest amount for the available balance. interest rate is fixed to 4.5%

- Create a Bank class to represent the banking system. Perform the following operation in main method:
  - create object for account class by calling parameter constructor.
  - deposit(amount: float): Deposit the specified amount into the account.
  - withdraw(amount: float): Withdraw the specified amount from the account.
  - calculate_interest(): Calculate and add interest to the account balance for savings accounts.

Task 8: Inheritance and polymorphism

1. Overload the deposit and withdraw methods in Account class as mentioned below.
   - deposit(amount: float): Deposit the specified amount into the account.
   - withdraw(amount: float): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.
   - deposit(amount: int): Deposit the specified amount into the account.
   - withdraw(amount: int): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.
   - deposit(amount: double): Deposit the specified amount into the account.
   - withdraw(amount: double): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.
2. Create Subclasses for Specific Account Types
   - Create subclasses for specific account types (e.g., `SavingsAccount`, `CurrentAccount`) that inherit from the `Account` class.
     - **SavingsAccount**: A savings account that includes an additional attribute for interest rate. **override** the calculate_interest() from Account class method to calculate interest based on the balance and interest rate.
     - **CurrentAccount**: A current account that includes an additional attribute overdraftLimit. A current account with no interest. Implement the withdraw() method to allow overdraft up to a certain limit (configure a constant for the overdraft limit).
3. Create a **Bank** class to represent the banking system. Perform the following operation in main method:
   - Display menu for user to create object for account class by calling parameter constructor. Menu should display options `SavingsAccount` and `CurrentAccount`. user can choose any one option to create account. use switch case for implementation.
   - **deposit(amount: float):** Deposit the specified amount into the account.
   - **withdraw(amount: float):** Withdraw the specified amount from the account. For saving account withdraw amount only if there is sufficient fund else display insufficient balance.

For Current Account withdraw limit can exceed the available balance and should not exceed the overdraft limit.

- **calculate_interest():** Calculate and add interest to the account balance for savings accounts.

Task 9: Abstraction

1. Create an abstract class BankAccount that represents a generic bank account. It should include the following attributes and methods:
    - Attributes:
        - Account number.
        - Customer name.
        - Balance.
    - Constructors:
        - Implement default constructors and overload the constructor with Account attributes, generate getter and setter, print all information of attribute methods for the attributes.
    - Abstract methods:
        - **deposit(amount: float):** Deposit the specified amount into the account.
        - **withdraw(amount: float):** Withdraw the specified amount from the account (implement error handling for insufficient funds).
        - **calculate_interest():** Abstract method for calculating interest.
2. Create two concrete classes that inherit from **BankAccount**:
    - **SavingsAccount**: A savings account that includes an additional attribute for interest rate. Implement the calculate_interest() method to calculate interest based on the balance and interest rate.
    - **CurrentAccount**: A current account with no interest. Implement the withdraw() method to allow overdraft up to a certain limit (configure a constant for the overdraft limit).
3. Create a Bank class to represent the banking system. Perform the following operation in main method:
    - Display menu for user to create object for account class by calling parameter constructor. Menu should display options \`SavingsAccount\` and \`CurrentAccount\`. user can choose any one option to create account. use switch case for implementation. create_account should display sub menu to choose type of accounts.
        - *Hint: Account acc = new SavingsAccount(); or Account acc = new CurrentAccount();*
    - deposit(amount: float): Deposit the specified amount into the account.
    - withdraw(amount: float): Withdraw the specified amount from the account. For saving account withdraw amount only if there is sufficient fund else display insufficient balance. For Current Account withdraw limit can exceed the available balance and should not exceed the overdraft limit.

- calculate_interest(): Calculate and add interest to the account balance for savings accounts.

Task 10: Has A Relation / Association

1. Create a `Customer` class with the following attributes:
    - Customer ID
    - First Name
    - Last Name
    - Email Address (validate with valid email address)
    - Phone Number (Validate 10-digit phone number)
    - Address
    - Methods and Constructor:
        - Implement default constructors and overload the constructor with Account attributes, generate getter, setter, print all information of attribute) methods for the attributes.
2. Create an `Account` class with the following attributes:
    - Account Number (a unique identifier).
    - Account Type (e.g., Savings, Current)
    - Account Balance
    - Customer (the customer who owns the account)
    - Methods and Constructor:
        - Implement default constructors and overload the constructor with Account attributes, generate getter, setter, (print all information of attribute) methods for the attributes.

Create a Bank Class and must have following requirements:

1. Create a Bank class to represent the banking system. It should have the following methods:
    - **create_account(Customer customer, long accNo, String accType, float balance)**: Create a new bank account for the given customer with the initial balance.
    - **get_account_balance(account_number: long)**: Retrieve the balance of an account given its account number. should return the current balance of account.
    - **deposit(account_number: long, amount: float)**: Deposit the specified amount into the account. Should return the current balance of account.
    - **withdraw(account_number: long, amount: float)**: Withdraw the specified amount from the account. Should return the current balance of account.
    - **transfer(from_account_number: long, to_account_number: int, amount: float)**: Transfer money from one account to another.
    - **getAccountDetails(account_number: long):** Should return the account and customer details.
2. Ensure that account numbers are automatically generated when an account is created, starting from 1001 and incrementing for each new account.

3. Create a BankApp class with a main method to simulate the banking system. Allow the user to interact with the system by entering commands such as "create_account", "deposit", "withdraw", "get_balance", "transfer", "getAccountDetails" and "exit." create_account should display sub menu to choose type of accounts and repeat this operation until user exit.

Task 11: Interface/abstract class, and Single Inheritance, static variable

1. Create a **'Customer'** class as mentioned above task.
2. Create an class '**Account'** that includes the following attributes. Generate account number using static variable.
    - Account Number (a unique identifier).
    - Account Type (e.g., Savings, Current)
    - Account Balance
    - Customer (the customer who owns the account)
    - lastAccNo
3. Create three child classes that inherit the Account class and each class must contain below mentioned attribute:
    - **SavingsAccount:** A savings account that includes an additional attribute for interest rate. Saving account should be created with minimum balance 500.
    - **CurrentAccount:** A Current account that includes an additional attribute for overdraftLimit(credit limit). withdraw() method to allow overdraft up to a certain limit. withdraw limit can exceed the available balance and should not exceed the overdraft limit.
    - **ZeroBalanceAccount**: ZeroBalanceAccount can be created with Zero balance.
4. Create **ICustomerServiceProvider** interface/abstract class with following functions:
    - **get_account_balance(account_number: long)**: Retrieve the balance of an account given its account number. should return the current balance of account.
    - **deposit(account_number: long, amount: float)**: Deposit the specified amount into the account. Should return the current balance of account.
    - **withdraw(account_number: long, amount: float)**: Withdraw the specified amount from the account. Should return the current balance of account. A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.
    - **transfer(from_account_number: long, to_account_number: int, amount: float)**: Transfer money from one account to another.
    - **getAccountDetails(account_number: long):** Should return the account and customer details.
5. Create **IBankServiceProvider** interface/abstract class with following functions:
    - **create_account(Customer customer, long accNo, String accType, float balance)**: Create a new bank account for the given customer with the initial balance.
    - **listAccounts()**:Account[] accounts: List all accounts in the bank.

- **calculateInterest():** the calculate_interest() method to calculate interest based on the balance and interest rate.

6. Create **CustomerServiceProviderImpl** class which implements I**CustomerServiceProvider** provide all implementation methods.
7. Create **BankServiceProviderImpl** class which inherits from **CustomerServiceProviderImpl and implements IBankServiceProvider**
    - Attributes
        - accountList: Array of **Accounts** to store any account objects.
        - branchName and branchAddress as String objects
8. Create **BankApp** class and perform following operation:
    - main method to simulate the banking system. Allow the user to interact with the system by entering choice from menu such as "create_account", "deposit", "withdraw", "get_balance", "transfer", "getAccountDetails", "ListAccounts" and "exit."
    - create_account should display sub menu to choose type of accounts and repeat this operation until user exit.
9. Place the interface/abstract class in service package and interface/abstract class implementation class, account class in bean package and Bank class in app package.
10. Should display appropriate message when the account number is not found and insufficient fund or any other wrong information provided.

Task 12: Exception Handling

throw the exception whenever needed and Handle in main method,

1. **InsufficientFundException** throw this exception when user try to withdraw amount or transfer amount to another account and the account runs out of money in the account.
2. **InvalidAccountException** throw this exception when user entered the invalid account number when tries to transfer amount, get account details classes.
3. **OverDraftLimitExcededException** thow this exception when current account customer try to with draw amount from the current account.
4. **NullPointerException** handle in main method**.**

Throw these exceptions from the methods in HMBank class. Make necessary changes to accommodate these exception in the source code. Handle all these exceptions from the main program.

Task 13: Collection

1. From the previous task change the **HMBank** attribute Accounts to List of Accounts and perform the same operation.
2. From the previous task change the **HMBank** attribute Accounts to Set of Accounts and perform the same operation.
    - Avoid adding duplicate Account object to the set.
    - Create Comparator<Account> object to sort the accounts based on customer name when listAccounts() method called.
3. From the previous task change the HMBank attribute Accounts to HashMap of Accounts and perform the same operation.

Task 14: Database Connectivity.

1. Create a **'Customer'** class as mentioned above task.
2. Create an class '**Account'** that includes the following attributes. Generate account number using static variable.
    - Account Number (a unique identifier).
    - Account Type (e.g., Savings, Current)
    - Account Balance
    - Customer (the customer who owns the account)
    - lastAccNo
3. Create a class **'TRANSACTION'** that include following attributes
    - Account
    - Description
    - Date and Time
    - TransactionType(Withdraw, Deposit, Transfer)
    - TransactionAmount
4. Create three child classes that inherit the Account class and each class must contain below mentioned attribute:
    - **SavingsAccount:** A savings account that includes an additional attribute for interest rate. Saving account should be created with minimum balance 500.
    - **CurrentAccount:** A Current account that includes an additional attribute for overdraftLimit(credit limit).
    - **ZeroBalanceAccount**: ZeroBalanceAccount can be created with Zero balance.
5. Create **ICustomerServiceProvider** interface/abstract class with following functions:
    - **get_account_balance(account_number: long)**: Retrieve the balance of an account given its account number. should return the current balance of account.
    - **deposit(account_number: long, amount: float)**: Deposit the specified amount into the account. Should return the current balance of account.
    - **withdraw(account_number: long, amount: float)**: Withdraw the specified amount from the account. Should return the current balance of account.
        - A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.
        - Current account customers are allowed withdraw overdraftLimit and available account balance. withdraw limit can exceed the available balance and should not exceed the overdraft limit.
    - **transfer(from_account_number: long, to_account_number: int, amount: float)**: Transfer money from one account to another. both account number should be validate from the database use getAccountDetails method.
    - **getAccountDetails(account_number: long):** Should return the account and customer details.

- **getTransations(account_number: long, FromDate:Date, ToDate: Date):** Should return the list of transaction between two dates.

6. Create **IBankServiceProvider** interface/abstract class with following functions:
    - **create_account(Customer customer, long accNo, String accType, float balance)**: Create a new bank account for the given customer with the initial balance.
    - **listAccounts()**: Array of BankAccount: List all accounts in the bank.(List[Account] accountsList)
    - **getAccountDetails(account_number: long):** Should return the account and customer details.
    - **calculateInterest():** the calculate_interest() method to calculate interest based on the balance and interest rate.
7. Create **CustomerServiceProviderImpl** class which implements I**CustomerServiceProvider** provide all implementation methods. These methods do not interact with database directly.
8. Create **BankServiceProviderImpl** class which inherits from **CustomerServiceProviderImpl and** implements **IBankServiceProvider.**
    - Attributes
        - accountList: List of **Accounts** to store any account objects.
        - transactionList: List of **Transaction** to store transaction objects.
        - branchName and branchAddress as String objects
9. Create I**BankRepository** interface/abstract class which include following methods to interact with database.
    - **createAccount(customer: Customer, accNo: long, accType: String, balance: float)**: Create a new bank account for the given customer with the initial balance and store in database.
    - **listAccounts()**: List<Account> accountsList: List all accounts in the bank from database.
    - **calculateInterest():** the calculate_interest() method to calculate interest based on the balance and interest rate.
    - **getAccountBalance(account_number: long)**: Retrieve the balance of an account given its account number. should return the current balance of account from database.
    - **deposit(account_number: long, amount: float)**: Deposit the specified amount into the account. Should update new balance in database and return the new balance.
    - **withdraw(account_number: long, amount: float)**: Withdraw amount should check the balance from account in database and new balance should updated in Database.
        - A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.
        - Current account customers are allowed withdraw overdraftLimit and available account balance. withdraw limit can exceed the available balance and should not exceed the overdraft limit.
    - **transfer(from_account_number: long, to_account_number: int, amount: float)**: Transfer money from one account to another. check the balance from account in database and balance should updated in Database.

- **getAccountDetails(account_number: long):** Should return the account and customer details from databse.
- **getTransations(account_number: long, FromDate:Date, ToDate: Date):** Should return the list of transaction between two dates from database.

10. Create **BankRepositoryImpl** class which implement the **IBankRepository** interface/abstract class and provide implementation of all methods and perform the database operations.
11. Create **DBUtil** class and add the following method.
    - **static getDBConn():Connection** Establish a connection to the database and return Connection reference
12. Create **BankApp** class and perform following operation:
    - main method to simulate the banking system. Allow the user to interact with the system by entering choice from menu such as "create_account", "deposit", "withdraw", "get_balance", "transfer", "getAccountDetails", "ListAccounts", "getTransactions" and "exit."
    - create_account should display sub menu to choose type of accounts and repeat this operation until user exit.
13. Place the interface/abstract class in service package and interface/abstract class implementation class, account class in bean package and Bank class in app package.
14. Should throw appropriate exception as mentioned in above task along with handle **SQLException**.