

Banking System Control Structure

Control statement :

Task 1: Conditional Statements

Task 2: Nested Conditional Statements

Task 3: Loop Structures

Task 4: Looping, Array and Data Validation T

ask 5: Password Validation

Task 6: Password Validation

OOPS, Collections and Exception Handling

Task 7: Class & Object

Task 8: Inheritance and polymorphism

Task 9: Abstraction

Task 10: Has A Relation / Association

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

Task 12: Exception Handling

Task 13: Collection Task

Task:14: Database Connectivity.

Task 1: Conditional Statements Banking System Control Structure

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.

Conditional Statements in Python

Python conditional statements direct the flow of execution depending on conditions. These conditions evaluate to True or False, and the program runs different blocks of code based on that.

Types of Conditional Statements

1. if Statement – Runs a block of code if the condition is True.
2. if-else Statement – Runs one block if the condition is True, else runs another block.
3. if-elif-else Statement – Tests multiple conditions one after another.
4. Nested if – An if nested within another if.

```

import mysql.connector
class LoanEligibilityChecker:
    def __init__(self):
        self.conn = mysql.connector.connect(
            host="localhost",
            user="root",
            password="123Password", # Replace with your MySQL password
            database="banking"
        )
        self.cursor = self.conn.cursor()

    def check_eligibility(self, name, credit_score, annual_income):
        eligible = credit_score > 700 and annual_income >= 50000

        # Insert the record into the database
        query = """
        INSERT INTO loan_eligibility (customer_name, credit_score, annual_income,
is_eligible)
        VALUES (%s, %s, %s, %s)
        """
        self.cursor.execute(query, (name, credit_score, annual_income, eligible))
        self.conn.commit()

        return eligible

    def close(self):
        self.cursor.close()
        self.conn.close()

if __name__ == "__main__":
    name = input("Enter Customer Name: ")
    try:
        credit_score = int(input("Enter Credit Score: "))
        annual_income = float(input("Enter Annual Income: "))

        checker = LoanEligibilityChecker()

```

```

if checker.check_eligibility(name, credit_score, annual_income):
    print("The customer is eligible for the loan.")
else:
    print("The customer is not eligible for the loan.")
checker.close()

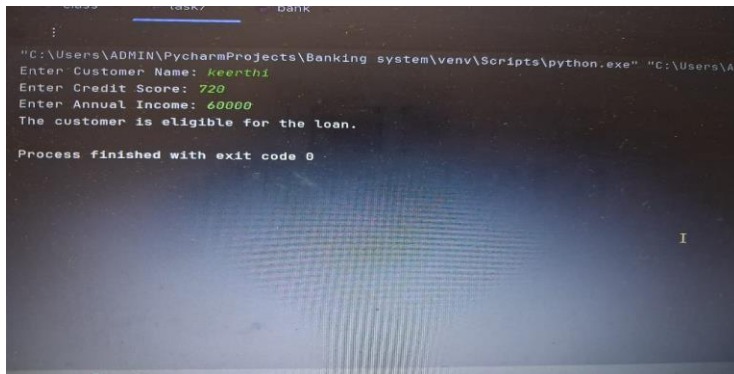
```

except ValueError:

```

    print("Invalid input! Please enter numeric values for credit score and income.")

```



```

"C:\Users\ADMIN\PycharmProjects\Banking system\venv\Scripts\python.exe" "C:\Users\AD
Enter Customer Name: keerthi
Enter Credit Score: 720
Enter Annual Income: 60000
The customer is eligible for the loan.
Process finished with exit code 0

```

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.

Nested Conditional Statement

A **nested conditional** means putting one if statement **inside** another if (or elif, else) block. It allows you to **check multiple levels of conditions**, and only proceed further if earlier conditions are true.

```

import mysql.connector

```

```

class ATM:

```

```

    def __init__(self):

```

```

        self.conn = mysql.connector.connect(

```

```

    host="localhost",
    user="root", # change if needed
    password="123Password", # your MySQL password
    database="banking" # your database name
)
self.cursor = self.conn.cursor()

def record_transaction(self, customer_id, transaction_type, amount):
    query = """
    INSERT INTO transactions (transaction_type, amount, customer_id)
    VALUES (%s, %s, %s)
    """
    self.cursor.execute(query, (transaction_type, amount, customer_id))
    self.conn.commit()

def atm_operations(self):
    customer_id = int(input("Enter your Customer ID: "))
    balance = float(input("Enter your Current Balance: "))

    print("\n--- ATM Menu ---")
    print("1. Check Balance")
    print("2. Withdraw")
    print("3. Deposit")

    choice = input("Choose an option (1-3): ")

    if choice == "1":
        print(f"Your current balance is: ${balance:.2f}")
    elif choice == "2":
        amount = float(input("Enter amount to withdraw: "))
        if amount > balance:
            print("Insufficient Balance!")
        else:
            if amount % 100 == 0 or amount % 500 == 0:
                balance -= amount
                print(f"Withdrawal Successful! New Balance: ${balance:.2f}")
                self.record_transaction(customer_id, "Withdraw", amount)
            else:

```

```

        print("Amount must be in multiples of 100 or 500.")
    elif choice == "3":
        amount = float(input("Enter amount to deposit: "))
        balance += amount
        print(f"Deposit Successful! New Balance: ${balance:.2f}")
        self.record_transaction(customer_id, "Deposit", amount)
    else:
        print("Invalid choice. Please try again.")

self.cursor.close()
self.conn.close()

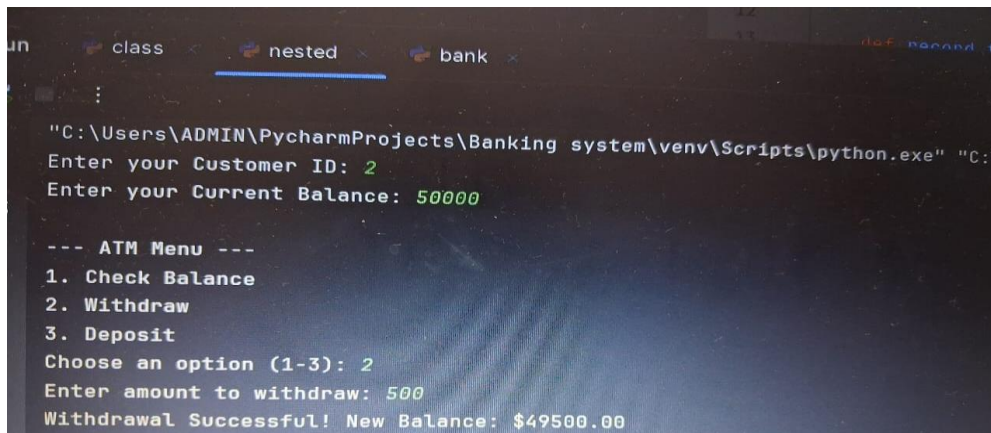
```

Run the ATM simulation

```

atm = ATM()
atm.atm_operations()

```



```

C:\Users\ADMIN\PycharmProjects\Banking system\venv\Scripts\python.exe "C:\
Enter your Customer ID: 2
Enter your Current Balance: 50000

--- ATM Menu ---
1. Check Balance
2. Withdraw
3. Deposit
Choose an option (1-3): 2
Enter amount to withdraw: 500
Withdrawal Successful! New Balance: $49500.00

```

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.

Looping in Python allows repetitive execution of code using structures like `for` and `while` loops. The `for` loop is ideal when the number of iterations is known, while `while` is used for indefinite repetition until a condition is false.

Data validation ensures user input or data is correct and safe before processing. It helps prevent errors by checking things like data type, value range, or format. Together, loops and validation help create robust programs that can repeatedly handle input correctly and reliably.

```
import mysql.connector
```

```
class CompoundInterestCalculator:
```

```
    def __init__(self):
```

```
        self.conn = mysql.connector.connect(
            host="localhost",
            user="root",
            password="123Password",
            database="banking"
```

```
        )
```

```
        self.cursor = self.conn.cursor()
```

```
        self.create_table()
```

```
    def create_table(self):
```

```
        query = """
```

```
        CREATE TABLE IF NOT EXISTS interest_calculations (
```

```
            id INT AUTO_INCREMENT PRIMARY KEY,
```

```
            customer_name VARCHAR(100),
```

```
            initial_balance FLOAT,
```

```
            annual_interest_rate FLOAT,
```

```
            years INT,
```

```
            future_balance FLOAT
```

```
        )
```

```

"""
self.cursor.execute(query)
self.conn.commit()

def calculate_and_store_interest(self, customer_name, balance, rate, years):
    future_balance = balance * ((1 + rate / 100) ** years)
    query = """
    INSERT INTO interest_calculations (customer_name, initial_balance,
annual_interest_rate, years, future_balance)
    VALUES (%s, %s, %s, %s, %s)
    """
    self.cursor.execute(query, (customer_name, balance, rate, years,
future_balance))
    self.conn.commit()
    return future_balance
def close(self):
    self.cursor.close()
    self.conn.close()

# ----- Main Program -----
if __name__ == "__main__":
    calc = CompoundInterestCalculator()
    num_customers = int(input("Enter number of customers: "))

    for i in range(num_customers):
        print(f"\nCustomer {i+1}")
        name = input("Enter Customer Name: ")
        initial_balance = float(input("Enter Initial Balance: "))
        annual_interest_rate = float(input("Enter Annual Interest Rate (%): "))
        years = int(input("Enter Number of Years: "))

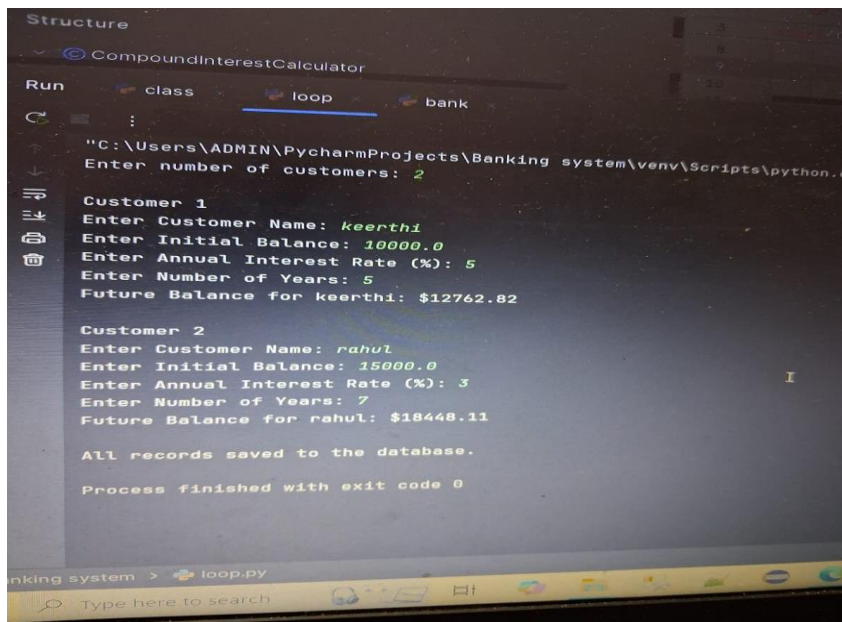
        future = calc.calculate_and_store_interest(name, initial_balance,
annual_interest_rate, years)
        print(f"Future Balance for {name}: ${future:.2f}")

    calc.close()

```



```
print("\nAll records saved to the database.")
```



```
Structure
CompoundInterestCalculator
Run
class
loop
bank
:
"C:\Users\ADMIN\PycharmProjects\Banking system\venv\Scripts\python.exe"
Enter number of customers: 2
Customer 1
Enter Customer Name: keerthi
Enter Initial Balance: 10000.0
Enter Annual Interest Rate (%): 5
Enter Number of Years: 5
Future Balance for keerthi: $12762.82
Customer 2
Enter Customer Name: rahul
Enter Initial Balance: 15000.0
Enter Annual Interest Rate (%): 3
Enter Number of Years: 7
Future Balance for rahul: $18448.11
All records saved to the database.
Process finished with exit code 0
```

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.

Password validation:

Password validation in Python ensures user credentials meet security standards. It uses if conditions to check for length (minimum 8 characters), at least one uppercase letter, and at least one digit. This helps protect user accounts from weak or easily guessable passwords.

```
import mysql.connector
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="123Password",
    database="banking"
)
```

```
cursor = conn.cursor()
```

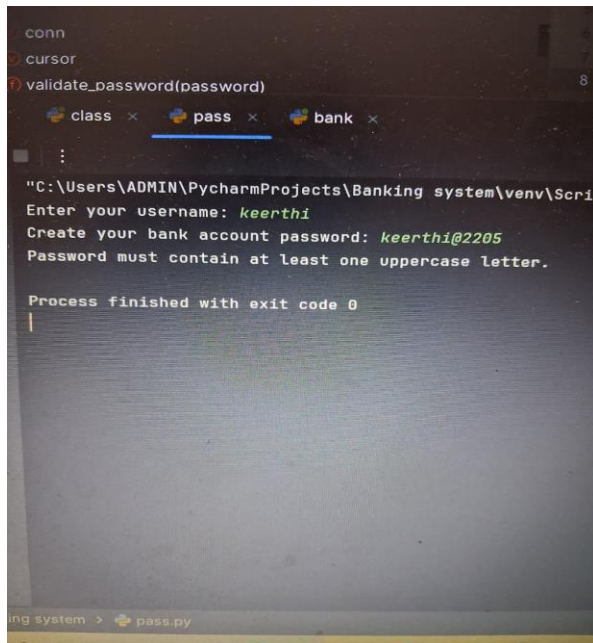
```
def validate_password(password):  
    if len(password) < 8:  
        return "Password must be at least 8 characters long.", False  
    if not any(char.isupper() for char in password):  
        return "Password must contain at least one uppercase letter.", False  
    if not any(char.isdigit() for char in password):  
        return "Password must contain at least one digit.", False  
    return "Password is valid!", True
```

```
username = input("Enter your username: ")  
user_password = input("Create your bank account password: ")
```

```
message, is_valid = validate_password(user_password)  
print(message)
```

```
if is_valid: insert_query = "INSERT INTO user_passwords (username, password)  
VALUES (%s, %s)"  
    cursor.execute(insert_query, (username, user_password))  
    conn.commit()  
    print("Password saved successfully in the database.")
```

```
cursor.close()  
conn.close()
```



```
conn
cursor
validate_password(password)

class x pass x bank x

"C:\Users\ADMIN\PycharmProjects\Banking system\venv\Scripts
Enter your username: keerthi
Create your bank account password: keerthi@2205
Password must contain at least one uppercase letter.

Process finished with exit code 0
```

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.

```
import mysql.connector
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="NewPassword",    # Replace with your actual password
    database="banking"        # Replace with your DB name
)
cursor = conn.cursor()

print("Welcome to HM Bank - Transaction Portal")
account_id = int(input("Enter your Account ID: "))while True:
    print("\nChoose an option:")
    print("1. Deposit")
    print("2. Withdraw")
    print("3. View Transaction History")
    print("4. Exit")
```

```

choice = input("Enter your choice: ")

if choice == "1":
    amount = float(input("Enter amount to deposit: "))
    cursor.execute(
        "INSERT INTO transactions (transaction_type, amount, account_id) VALUES
(%s, %s, %s)",
        ("Withdraw", amount, account_id)
    )
    conn.commit()
    print(f"Deposited ₹{amount} successfully.")

elif choice == "2":
    amount = float(input("Enter amount to withdraw: "))
    cursor.execute(
        "INSERT INTO transactions (transaction_type, amount, account_id) VALUES
(%s, %s, %s)",
        ("Withdraw", amount, account_id)
    )
    conn.commit()
    print(f"Withdrawn ₹{amount} successfully.")

elif choice == "3":
    print(f"\nTransaction history for Account ID {account_id}:")
    cursor.execute(
        "INSERT INTO transactions (transaction_type, amount, account_id) VALUES
(%s, %s, %s)",
        ("Withdraw", amount, account_id)
    )
    transactions = cursor.fetchall()
    if transactions:
        for txn in transactions:
            print(f"ID: {txn[0]}, Type: {txn[1]}, Amount: ₹{txn[2]}")
    else:
        print("No transactions found.")

elif choice == "4":

```

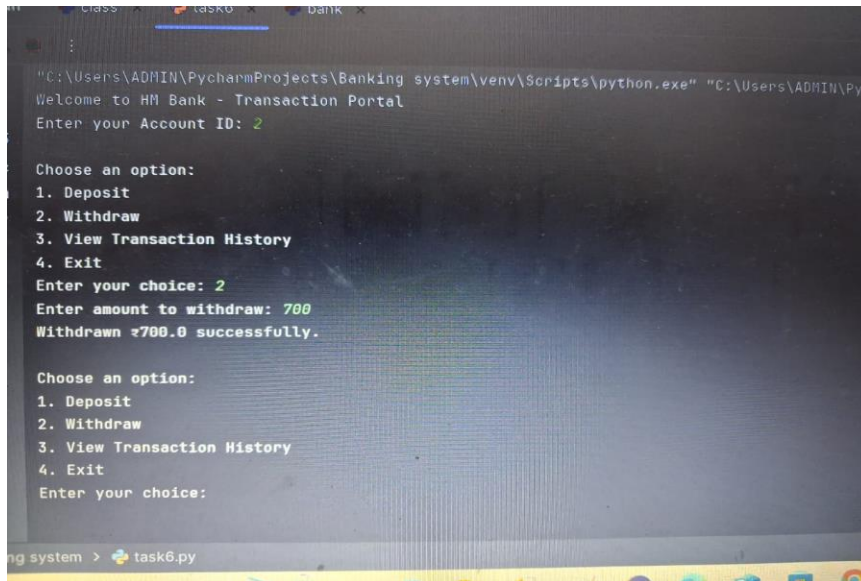
```
print("Exiting... Thank you for using HM Bank!")  
break
```

else:

```
print("Invalid choice. Please try again.")
```

```
cursor.close()
```

```
conn.close()
```



```
"C:\Users\ADMIN\PycharmProjects\Banking system\venv\Scripts\python.exe" "C:\Users\ADMIN\Py  
Welcome to HM Bank - Transaction Portal  
Enter your Account ID: 2  
  
Choose an option:  
1. Deposit  
2. Withdraw  
3. View Transaction History  
4. Exit  
Enter your choice: 2  
Enter amount to withdraw: 700  
Withdrawn ₹700.0 successfully.  
  
Choose an option:  
1. Deposit  
2. Withdraw  
3. View Transaction History  
4. Exit  
Enter your choice:
```

OOPS, Collections and Exception Handling

Task 7: Class & Object

1. Create a `Customer` class with the following confidential attributes:

- Attributes

Customer ID

First Name o

Last Name o

Email Address

- o Phone Number

- o Address

- Constructor and Methods

- o 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
- o Account Number

- o Account Type (e.g., Savings, Current)

- o Account Balance

- Constructor and Methods

- o Implement default constructors and overload the constructor with Account attributes,

- o Generate getter and setter, (print all information of attribute) methods for the attributes.

- o 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:

- o create object for account class by calling parameter constructor
- . o deposit(amount: float): Deposit the specified amount into the account.
- o withdraw(amount: float): Withdraw the specified amount from the account.
- o calculate_interest(): Calculate and add interest to the account balance for savings accounts.

Class:

A **class** is a **blueprint** or **template** used to create objects.

It defines **attributes (data)** and **methods (functions)** that the objects will have.

Object:

An **object** is an **instance** of a class.

It represents a **real-world entity** created using the class blueprint.

```
import mysql.connector
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="NewPassword",
    database="banking"
)
cursor = conn.cursor()

class Customer:
    def __init__(self, customer_id=None,
first_name=None,last_name=None,email=None,phone=None,address=None):self.cst
omer_id = customer_id
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone = phone
        self.address = address
    def display_customer_info(self):
        print(
```

```
f"Customer ID: {self.customer_id}\nFirst Name: {self.first_name}\nLast Name:
{self.last_name}\nEmail: {self.email}\nPhone: {self.phone}\nAddress:
{self.address}\n")
```

```
class Account:
    def __init__(self, account_id=None, account_type=None, balance=0.0):
        self.account_id = account_id
        self.account_type = account_type
        self.balance = balance
```

```
    def save_to_db(self):
```

```
        cursor.execute("UPDATE accounts SET balance = %s WHERE account_id = %s",
                        (self.balance, self.account_id))
        conn.commit()
```

```
    def deposit(self, amount):
```

```
        self.balance += amount
        self.save_to_db()
        print(f"Deposited {amount}. New Balance: {self.balance}")
```

```
    def withdraw(self, amount):
```

```
        if self.balance >= amount:
            self.balance -= amount
            self.save_to_db()
            print(f"Withdrawn {amount}. New Balance: {self.balance}")
        else:
            print("Insufficient balance!")
```

```
    def calculate_interest(self):
```

```
        if self.account_type.lower() == "savings":
            interest = self.balance * 0.045 # 4.5% interest
            self.balance += interest
            self.save_to_db()
            print(f"Interest added: {interest}. New Balance: {self.balance}")
        else:
```



```

        print("Interest calculation is only available for savings accounts.")

    def display_account_info(self):
        print(f"Account ID: {self.account_id}\nAccount Type: {self.account_type}\nBalance: {self.balance}\n")

class Bank:

    def create_account():
        """Creates a new bank account and stores it in the database."""
        acc_type = input("Enter Account Type (Savings/Current): ")
        balance = float(input("Enter Initial Balance: "))

        cursor.execute("INSERT INTO accounts (account_type, balance) VALUES (%s, %s)", (acc_type, balance))
        conn.commit()

        account_id = cursor.lastrowid # Get the last inserted ID
        print(f"Account Created Successfully! Account ID: {account_id}")
        return Account(account_id, acc_type, balance)

    def perform_operations():

        account_id = input("Enter Account ID: ")

        cursor.execute("SELECT account_type, balance FROM accounts WHERE account_id = %s", (account_id,))
        result = cursor.fetchone()

        if result:
            acc_type, balance = result
            account = Account(account_id, acc_type, balance)

        while True:
            print("\nBanking System Menu:")
            print("1. Deposit\n2. Withdraw\n3. Calculate Interest\n4. Exit")

```

```

    choice = input("Enter your choice: ")

    if choice == "1":
        amount = float(input("Enter amount to deposit: "))
        account.deposit(amount)
    elif choice == "2":
        amount = float(input("Enter amount to withdraw: "))
        account.withdraw(amount)
    elif choice == "3":
        account.calculate_interest()
    elif choice == "4":
        print("Exiting...")
        break
    else:
        print("Invalid option! Please try again.")
else:
    print("Account not found!")

if __name__ == "__main__":
    while True:
        print("\nBanking System Main Menu:")
        print("1. Create Account\n2. Perform Account Operations\n3. Exit")

        option = input("Enter your choice: ")

        if option == "1":
            Bank.create_account()
        elif option == "2":
            Bank.perform_operations()
        elif option == "3":
            print("Exiting System...")
            break
        else:
            print("Invalid Choice! Please enter 1, 2, or 3.")

    cursor.close()
    conn.close()

```

```
class task0 bank
{
}

"C:\Users\ADMIN\PycharmProjects\Banking system\venv\Scripts\python.exe" "C:\Users\ADMIN\PycharmProjects\Banking system\main.py"

Banking System Main Menu:
1. Create Account
2. Perform Account Operations
3. Exit
Enter your choice: 2
Enter Account ID: 1

Banking System Menu:
1. Deposit
2. Withdraw
3. Calculate Interest
4. Exit
Enter your choice: 3
Interest added: 45.0. New Balance: 1045.0

Banking System Menu:
1. Deposit
2. Withdraw
3. Calculate Interest
4. Exit
Enter your choice:
```

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. o 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.

o **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. . All rights 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.

Inheritance:

Inheritance allows a class (child class) to **inherit** attributes and methods from another class (parent class). It promotes **code reusability** and allows hierarchical classification of classes.

Types of Inheritance in Python:

1. **Single Inheritance** – Child class inherits from one parent class.
2. **Multiple Inheritance** – Child class inherits from multiple parent classes.
3. **Multilevel Inheritance** – A class inherits from another child class.
4. **Hierarchical Inheritance** – Multiple child classes inherit from a single parent class.
5. **Hybrid Inheritance** – A combination of different types of inheritance.

Polymorphism:

Polymorphism means "many forms" and allows different classes to have methods with the **same name but different behaviors**.

Types of Polymorphism in Python

1. **Method Overriding** (Dynamic Polymorphism)
 - a. A child class **redefines** a method from the parent class with different functionality.
2. **Method Overloading** (Static Polymorphism - Not natively supported in Python)
 - a. The same method name can accept different types of arguments (handled using *args or isinstance()).

We can apply these concepts to a **banking system**:

- **Inheritance:** SavingsAccount and CurrentAccount inherit from Account
- **Polymorphism:**
 - withdraw() behaves **differently** for Savings and Current accounts.
 - calculate_interest() is overridden in SavingsAccount.

```
import mysql.connector
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="NewPassword",
    database="banking"
)
cursor = conn.cursor()

class Account:
    def __init__(self, account_id, account_type, balance=0.0):
        self.account_id = account_id
        self.account_type = account_type
```

```

        self.balance = float(balance) # Ensuring balance is a float
def save_to_db(self):

    cursor.execute("UPDATE accounts SET balance = %s WHERE account_id = %s",
(self.balance, self.account_id))
    conn.commit()

def deposit(self, amount):
    if isinstance(amount, (int, float)):
        self.balance += float(amount)
        self.save_to_db()
        print(f"Deposited {amount}. New Balance: {self.balance}")
    else:
        print("Invalid amount. Please enter a number.")

def withdraw(self, amount):
    if isinstance(amount, (int, float)):
        if self.balance >= float(amount):
            self.balance -= float(amount)
            self.save_to_db()
            print(f"Withdrawn {amount}. New Balance: {self.balance}")
        else:
            print("Insufficient balance!")
    else:
        print("Invalid amount. Please enter a number.")

def display_account_info(self):
    print(f"Account ID: {self.account_id}\nAccount Type:
{self.account_type}\nBalance: {self.balance}\n")

class SavingsAccount(Account):
    INTEREST_RATE = 0.045 # 4.5% Interest

    def __init__(self, account_id, balance=0.0):
        super().__init__(account_id, "Savings", balance)

    def calculate_interest(self):
        interest = self.balance * self.INTEREST_RATE

```

```
self.balance += interest
self.save_to_db()
print(f"Interest added: {interest}. New Balance: {self.balance}")
```

```
class CurrentAccount(Account):
```

```
    OVERDRAFT_LIMIT = 5000.0
```

```
    def __init__(self, account_id, balance=0.0):
        super().__init__(account_id, "Current", balance)
```

```
    def withdraw(self, amount):
```

```
        if isinstance(amount, (int, float)):
```

```
            if self.balance - float(amount) >= -self.OVERDRAFT_LIMIT:
```

```
                self.balance -= float(amount)
```

```
                self.save_to_db()
```

```
                print(f"Withdrawn {amount}. New Balance: {self.balance}")
```

```
            else:
```

```
                print(f"Overdraft limit exceeded! You can withdraw up to
{self.OVERDRAFT_LIMIT} beyond balance.")
```

```
            else:
```

```
                print("Invalid amount. Please enter a number.")
```

```
class Bank:
```

```
    @staticmethod
```

```
    def create_account():
```

```
        """Creates an account based on user choice (Savings or Current)."""
```

```
        print("Choose Account Type:\n1. Savings Account\n2. Current Account")
```

```
        choice = input("Enter choice (1 or 2): ")
```

```
        balance = float(input("Enter Initial Balance: "))
```

```
        cursor.execute("INSERT INTO accounts (account_type, balance) VALUES
(%s, %s)",
```

```
            ("Savings" if choice == "1" else "Current", balance))
```

```
        conn.commit()
```

```
        account_id = cursor.lastrowid
```

```
        if choice == "1":
```

```
            account = SavingsAccount(account_id, balance)
```

```

elif choice == "2":
    account = CurrentAccount(account_id, balance)
else:
    print("Invalid choice!")
    return None

print(f"Account Created Successfully! Account ID: {account_id}")
return account

@staticmethod
def perform_operations():
    account_id = input("Enter Account ID: ")

    cursor.execute("SELECT account_type, balance FROM accounts WHERE
account_id = %s", (account_id,))
    result = cursor.fetchone()

    if result:
        acc_type, balance = result
        account = SavingsAccount(account_id, balance) if acc_type == "Savings" else
CurrentAccount(account_id, l.connector
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="NewPassword",
    database="banking"
)
cursor = conn.cursor()

class Account:
    def __init__(self, account_id, account_type, balance=0.0):
        self.account_id = account_id
        self.account_type = account_type
        self.balance = float(balance) # Ensuring balance is a float

    def save_to_db(self):

        cursor.execute("UPDATE accounts SET balance = %s WHERE account_id = %s",

```



```
(self.balance, self.account_id))  
    conn.commit()
```

```
def deposit(self, amount):  
    if isinstance(amount, (int, float)):  
        self.balance += float(amount)  
        self.save_to_db()  
        print(f"Deposited {amount}. New Balance: {self.balance}")  
    else:  
        print("Invalid amount. Please enter a number.")
```

```
def withdraw(self, amount):  
    if isinstance(amount, (int, float)):  
        if self.balance >= float(amount):  
            self.balance -= float(amount)  
            self.save_to_db()  
            print(f"Withdrawn {amount}. New Balance: {self.balance}")  
        else:  
            print("Insufficient balance!")  
    else:  
        print("Invalid amount. Please enter a number.")
```

```
def display_account_info(self):  
    print(f"Account ID: {self.account_id}\nAccount Type:  
{self.account_type}\nBalance: {self.balance}\n")
```

```
class SavingsAccount(Account):  
    INTEREST_RATE = 0.045 # 4.5% Interest
```

```
def __init__(self, account_id, balance=0.0):  
    super().__init__(account_id, "Savings", balance)
```

```
def calculate_interest(self):  
    interest = self.balance * self.INTEREST_RATE  
    self.balance += interest  
    self.save_to_db()  
    print(f"Interest added: {interest}. New Balance: {self.balance}")
```

```

class CurrentAccount(Account):
    OVERDRAFT_LIMIT = 5000.0

    def __init__(self, account_id, balance=0.0):
        super().__init__(account_id, "Current", balance)

    def withdraw(self, amount):
        if isinstance(amount, (int, float)):
            if self.balance - float(amount) >= -self.OVERDRAFT_LIMIT:
                self.balance -= float(amount)
                self.save_to_db()
                print(f"Withdrawn {amount}. New Balance: {self.balance}")
            else:
                print(f"Overdraft limit exceeded! You can withdraw up to {self.OVERDRAFT_LIMIT} beyond balance.")
        else:
            print("Invalid amount. Please enter a number.")

class Bank:
    @staticmethod
    def create_account():
        """Creates an account based on user choice (Savings or Current)."""
        print("Choose Account Type:\n1. Savings Account\n2. Current Account")
        choice = input("Enter choice (1 or 2): ")
        balance = float(input("Enter Initial Balance: "))

        cursor.execute("INSERT INTO accounts (account_type, balance) VALUES (%s, %s)",
            ("Savings" if choice == "1" else "Current", balance))
        conn.commit()
        account_id = cursor.lastrowid

        if choice == "1":
            account = SavingsAccount(account_id, balance)
        elif choice == "2":
            account = CurrentAccount(account_id, balance)
        else:
            print("Invalid choice!")

```

```

    return None

    print(f"Account Created Successfully! Account ID: {account_id}")
    return account

    @staticmethod
    def perform_operations():
        account_id = input("Enter Account ID: ")

        cursor.execute("SELECT account_type, balance FROM accounts WHERE
account_id = %s", (account_id,))
        result = cursor.fetchone()

        if result:
            acc_type, balance = result
            account = SavingsAccount(account_id, balance) if acc_type == "Savings" else
CurrentAccount(account_id,
                                balance)

        while True:
            print("\nBanking System Menu:")
            print("1. Deposit\n2. Withdraw\n3. Calculate Interest (Savings Only)\n4. Exit")
            choice = input("Enter your choice: ")

            if choice == "1":
                amount = float(input("Enter amount to deposit: "))
                account.deposit(amount)
            elif choice == "2":
                amount = float(input("Enter amount to withdraw: "))
                account.withdraw(amount)
            elif choice == "3" and isinstance(account, SavingsAccount):
                account.calculate_interest()
            elif choice == "4":
                print("Exiting...")
                break
            else:
                print("Invalid option! Please try again.")
        else:

```

```

    print("Account not found!")

if __name__ == "__main__":
    while True:
        print("\nBanking System Main Menu:")
        print("1. Create Account\n2. Perform Account Operations\n3. Exit")

        option = input("Enter your choice: ")

        if option == "1":
            Bank.create_account()
        elif option == "2":
            Bank.perform_operations()
        elif option == "3":
            print("Exiting System...")
            break
        else:
            print("Invalid Choice! Please enter 1, 2, or 3.")
    cursor.close()
    conn.close()

while True:
    print("\nBanking System Menu:")
    print("1. Deposit\n2. Withdraw\n3. Calculate Interest (Savings Only)\n4. Exit")
    choice = input("Enter your choice: ")

    if choice == "1":
        amount = float(input("Enter amount to deposit: "))
        account.deposit(amount)
    elif choice == "2":
        amount = float(input("Enter amount to withdraw: "))
        account.withdraw(amount)
    elif choice == "3" and isinstance(account, SavingsAccount):
        account.calculate_interest()
    elif choice == "4":
        print("Exiting...")

```

```
        break
    else:
        print("Invalid option! Please try again.")
    else:
        print("Account not found!")

if __name__ == "__main__":
    while True:
        print("\nBanking System Main Menu:")
        print("1. Create Account\n2. Perform Account Operations\n3. Exit")

        option = input("Enter your choice: ")

        if option == "1":
            Bank.create_account()
        elif option == "2":
            Bank.perform_operations()
        elif option == "3":
            print("Exiting System...")
            break
        else:
            print("Invalid Choice! Please enter 1, 2, or 3.")
    cursor.close()
    conn.close()
```

```
"C:\Users\ADMIN\PycharmProjects\Banking system\venv\Scripts\python.exe" "C:\Users\ADMIN\PycharmProjects\Banking system\inheritance.py"

Banking System Main Menu:
1. Create Account
2. Perform Account Operations
3. Exit
Enter your choice: 1
Choose Account Type:
1. Savings Account
2. Current Account
Enter choice (1 or 2): 1
Enter Initial Balance: 5000
Account Created Successfully! Account ID: 12

Banking System Main Menu:
1. Create Account
2. Perform Account Operations
3. Exit
Enter your choice: 2
Enter Account ID: 1

Banking System Menu:
1. Deposit
2. Withdraw
3. Calculate Interest (Savings Only)
4. Exit
Enter your choice: 2
Enter amount to withdraw: 550
Withdrawn 550.0. New Balance: 495.0
```

Task 9: Abstraction

1. Create an abstract class BankAccount that represents a generic bank account. It should include the following attributes and methods:

- Attributes:

- o Account number.
- o Customer name.
- o Balance.

- Constructors:

- o 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:

o deposit(amount: float): Deposit the specified amount into the account. o withdraw(amount: float): Withdraw the specified amount from the account (implement error handling for insufficient funds).

o 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. o 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.

Abstraction:

Abstraction is a core concept in **Object-Oriented Programming (OOP)** that allows you to **hide complex implementation details** and show only the **necessary parts** of an object to the user.

How it is used in Banking system:

Abstraction- With bank account and Abstract class

Abstract method- deposit, withdraw, calculate interest

Implementation- Provided in savings account and current account

```
from abc import ABC, abstractmethod  
import mysql.connector
```

```
conn = mysql.connector.connect(  
    host="localhost",  
    user="root",  
    password="NewPassword",  
    database="banking"  
)  
cursor = conn.cursor()
```

```
class BankAccount(ABC):
```

```
    def __init__(self, account_number, customer_id, balance):  
        self.account_number = account_number  
        self.customer_id = customer_id  
        self.balance = balance
```

```
    def get_account_number(self):  
        return self.account_number
```

```
    def get_customer_id(self):  
        return self.customer_id
```

```
    def get_balance(self):  
        return self.balance
```

```
    def set_balance(self, new_balance):  
        self.balance = new_balance
```

```
    def display_account_info(self):  
        first_name = input("Enter Customer First Name: ")  
        last_name = input("Enter Customer Last Name: ")  
        dob = input("Enter Date of Birth (YYYY-MM-DD): ") # Format matters!
```

```
    cursor.execute(  
        "INSERT INTO customers (first_name, last_name, DOB) VALUES (%s, %s, %s)",
```



```
(first_name, last_name, dob)
)
conn.commit()
```

```
customer_name = cursor.fetchone()[0]
print(f"\nAccount Number: {self.account_number}")
print(f"Customer Name: {customer_name}")
print(f"Balance: {self.balance}\n")
```

```
def deposit(self, amount):
    pass
def withdraw(self, amount):
    pass
```

```
def calculate_interest(self):
    pass
```

```
class SavingsAccount(BankAccount):
    INTEREST_RATE = 0.045
```

```
def __init__(self, account_number, customer_id, balance):
    super().__init__(account_number, customer_id, balance)
```

```
def deposit(self, amount):
    self.balance += amount
    first_name = input("Enter Customer First Name: ")
    last_name = input("Enter Customer Last Name: ")
    dob = input("Enter Date of Birth (YYYY-MM-DD): ") # Format matters!
```

```
cursor.execute(
    "INSERT INTO customers (first_name, last_name, DOB) VALUES (%s, %s, %s)",
    (first_name, last_name, dob)
)
conn.commit()
```

```
print(f"Deposited {amount}. New Balance: {self.balance}")
```

```

def withdraw(self, amount):
    if self.balance >= amount:
        self.balance -= amount
        first_name = input("Enter Customer First Name: ")
        last_name = input("Enter Customer Last Name: ")
        dob = input("Enter Date of Birth (YYYY-MM-DD): ") # Format matters!

        cursor.execute(
            "INSERT INTO customers (first_name, last_name, DOB) VALUES (%s, %s, %s)",
            (first_name, last_name, dob)
        )
        conn.commit()

        print(f"Withdrawn {amount}. New Balance: {self.balance}")
    else:
        print("Insufficient balance!")

```

```

def calculate_interest(self):
    interest = self.balance * SavingsAccount.INTEREST_RATE
    self.balance += interest
    first_name = input("Enter Customer First Name: ")
    last_name = input("Enter Customer Last Name: ")
    dob = input("Enter Date of Birth (YYYY-MM-DD): ") # Format matters!

    cursor.execute(
        "INSERT INTO customers (first_name, last_name, DOB) VALUES (%s, %s, %s)",
        (first_name, last_name, dob)
    )
    conn.commit()

    print(f"Interest added: {interest}. New Balance: {self.balance}")

```

```

class CurrentAccount(BankAccount):
    OVERDRAFT_LIMIT = 5000

    def __init__(self, account_number, customer_id, balance):
        super().__init__(account_number, customer_id, balance)

```

```

def deposit(self, amount):
    self.balance += amount
    first_name = input("Enter Customer First Name: ")
    last_name = input("Enter Customer Last Name: ")
    dob = input("Enter Date of Birth (YYYY-MM-DD): ") # Format matters!

    cursor.execute(
        "INSERT INTO customers (first_name, last_name, DOB) VALUES (%s, %s, %s)",
        (first_name, last_name, dob)
    )
    conn.commit()

    print(f"Deposited {amount}. New balance: {self.balance}")

def withdraw(self, amount):
    if self.balance - amount >= -CurrentAccount.OVERDRAFT_LIMIT:
        self.balance -= amount
        first_name = input("Enter Customer First Name: ")
        last_name = input("Enter Customer Last Name: ")
        dob = input("Enter Date of Birth (YYYY-MM-DD): ") # Format matters!

        cursor.execute(
            "INSERT INTO customers (first_name, last_name, DOB) VALUES (%s, %s, %s)",
            (first_name, last_name, dob)
        )
        conn.commit()

        print(f"Withdrawn {amount}. New Balance: {self.balance}")
    else:
        print("Withdrawal denied! Overdraft limit exceeded.")

def calculate_interest(self):
    print("No interest for Current Accounts.")

class Bank:

    def create_account():
        print("\nChoose Account Type:")

```

```
print("1. Savings Account")
print("2. Current Account")
choice = input("Enter choice (1 or 2): ")
```

```
first_name = input("Enter Customer First Name: ")
last_name = input("Enter Customer Last Name: ")
dob = input("Enter Date of Birth (YYYY-MM-DD): ")
balance = float(input("Enter Initial Balance: "))
```

```
first_name = input("Enter Customer First Name: ")
last_name = input("Enter Customer Last Name: ")
dob = input("Enter Date of Birth (YYYY-MM-DD): ") # Format matters!
```

```
cursor.execute(
    "INSERT INTO customers (first_name, last_name, DOB) VALUES (%s, %s, %s)",
    (first_name, last_name, dob)
)
conn.commit()
```

```
cursor.execute("SELECT LAST_INSERT_ID()")
customer_id = cursor.fetchone()[0]
```

```
first_name = input("Enter Customer First Name: ")
last_name = input("Enter Customer Last Name: ")
dob = input("Enter Date of Birth (YYYY-MM-DD): ") # Format matters!
```

```
cursor.execute(
    "INSERT INTO customers (first_name, last_name, DOB) VALUES (%s, %s, %s)",
    (first_name, last_name, dob)
)
conn.commit()
```

```
acc_type = "Savings" if choice == "1" else "Current"
```

```
first_name = input("Enter Customer First Name: ")
last_name = input("Enter Customer Last Name: ")
```

```

first_name = input("Enter Customer First Name: ")
last_name = input("Enter Customer Last Name: ")
dob = input("Enter Date of Birth (YYYY-MM-DD): ") # Format matters!

cursor.execute(
    "INSERT INTO customers (first_name, last_name, DOB) VALUES (%s, %s, %s)",
    (first_name, last_name, dob)
)
conn.commit()

print(f"Account created successfully! Your Account ID: {acc_num}")

if choice == "1":
    return SavingsAccount(acc_num, customer_id, balance)
else:
    return CurrentAccount(acc_num, customer_id, balance)

def perform_operations(account):
    while True:
        print("\nBanking System Menu:")
        print("1. Deposit\n2. Withdraw\n3. Calculate Interest\n4. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            amount = float(input("Enter amount to deposit: "))
            account.deposit(amount)
        elif choice == 2:
            amount = float(input("Enter amount to withdraw: "))
            account.withdraw(amount)
        elif choice == 3:
            account.calculate_interest()
        elif choice == 4:
            print("Exiting...")
            break
        else:
            print("Invalid option!")

```

```

if __name__ == "__main__":
    while True:
        print("\nBanking System Main Menu:")
        print("1. Create Account\n2. Perform Account Operations\n3. Exit")
        option = int(input("Enter your choice: "))

        if option == 1:
            account = Bank.create_account()
            if account:
                print("Account Created Successfully!")
                account.display_account_info()
        elif option == 2:
            acc_num = input("Enter Account Number: ")
            first_name = input("Enter Customer First Name: ")
            last_name = input("Enter Customer Last Name: ")
            dob = input("Enter Date of Birth (YYYY-MM-DD): ") # Format matters!

            cursor.execute(
                "INSERT INTO customers (first_name, last_name, DOB) VALUES (%s, %s, %s)",
                (first_name, last_name, dob)
            )
            conn.commit()

            result = cursor.fetchone()

            if result:
                customer_id, acc_type, balance = result
                if acc_type == "Savings":
                    account = SavingsAccount(acc_num, customer_id, balance)
                else:
                    account = CurrentAccount(acc_num, customer_id, balance)

                Bank.perform_operations(account)
            else:
                print("Account not found!")

        elif option == 3:
            print("Exiting System...")

```

```

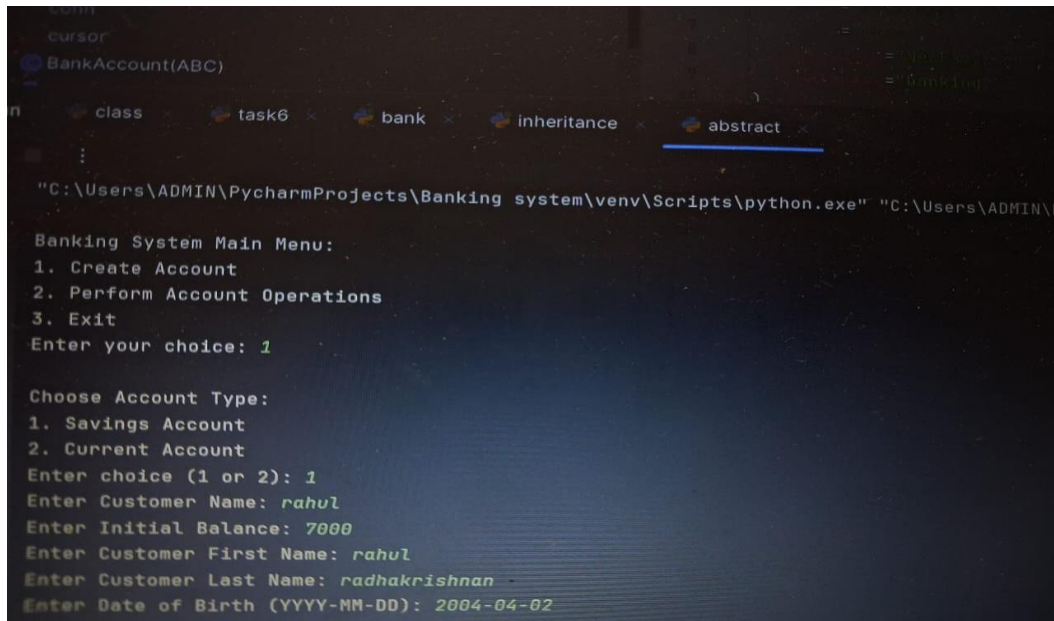
        break
    else:
        print("Invalid Choice!")

```

```

cursor.close()
conn.close()

```



The screenshot shows a terminal window with a dark background. At the top, there are several tabs: 'cursor', 'task6', 'bank', 'inheritance', and 'abstract'. The 'abstract' tab is currently selected. Below the tabs, the terminal displays the following text:

```

"C:\Users\ADMIN\PycharmProjects\Banking system\venv\Scripts\python.exe" "C:\Users\ADMIN\Py
Banking System Main Menu:
1. Create Account
2. Perform Account Operations
3. Exit
Enter your choice: 1

Choose Account Type:
1. Savings Account
2. Current Account
Enter choice (1 or 2): 1
Enter Customer Name: rahul
Enter Initial Balance: 7000
Enter Customer First Name: rahul
Enter Customer Last Name: radhakrishnan
Enter Date of Birth (YYYY-MM-DD): 2004-04-02

```

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: o 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: o 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.

Has A Relationship/Association:

a **Has-A relationship** (also known as **association**) is when **one class contains a reference to another class**. This means that an object of one class is used as an attribute in another class.

It's a key principle of **object-oriented programming** (OOP) and helps in creating modular, reusable, and maintainable code.

- Bank *has* multiple Account objects.
- Each Account *has* a Customer object.

```
import mysql.connector
import re
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="NewPassword",
    database="banking"
)
cursor = conn.cursor()

class Customer:
    def __init__(self, first_name, last_name, email, phone_number, address):
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone_number = phone_number
        self.address = address

    def is_valid_email(self):
        return re.match(r"^[^@]+@[^@]+\.[^@]+", self.email)

    def is_valid_phone(self):
        return re.match(r"^\d{10}$", self.phone_number)

class Account:
    def __init__(self, account_type, balance, customer_id):
        self.account_type = account_type
        self.balance = balance
```

```
self.customer_id = customer_id
```

```
class Bank:
```

```
def create_account(self):
```

```
    print("\n--- Enter Customer Information ---")
```

```
    first_name = input("First Name: ")
```

```
    last_name = input("Last Name: ")
```

```
    email = input("Email: ")
```

```
    phone = input("Phone (10 digits): ")
```

```
    address = input("Address: ")
```

```
    customer = Customer(first_name, last_name, email, phone, address)
```

```
    if not customer.is_valid_email() or not customer.is_valid_phone():
```

```
        print("Invalid email or phone number format.")
```

```
        return
```

```
    cursor.execute("""
```

```
        INSERT INTO customers (first_name, last_name, email, phone_number, address)
```

```
        VALUES (%s, %s, %s, %s, %s)
```

```
    """, (first_name, last_name, email, phone, address))
```

```
    conn.commit()
```

```
    customer_id = cursor.lastrowid
```

```
    print("\n--- Enter Account Details ---")
```

```
    acc_type = input("Account Type (Savings/Current): ")
```

```
    balance = float(input("Initial Balance: "))
```

```
    cursor.execute("""
```

```
        INSERT INTO accounts (account_type, balance, customer_id)
```

```
        VALUES (%s, %s, %s)
```

```
    """, (acc_type, balance, customer_id))
```

```
    conn.commit()
```

```
    print(" Account successfully created!\n")
```

```
def get_account_balance(self, acc_id):
```

```
    cursor.execute("SELECT balance FROM accounts WHERE account_id = %s",
```

```

(acc_id,))
    result = cursor.fetchone()
    if result:
        print(f"Current Balance: ₹{result[0]}")
    else:
        print(" Account not found.")

def deposit(self, acc_id, amount):
    cursor.execute("UPDATE accounts SET balance = balance + %s WHERE account_id
= %s", (amount, acc_id))
    conn.commit()
    self.get_account_balance(acc_id)

def withdraw(self, acc_id, amount):
    cursor.execute("SELECT balance FROM accounts WHERE account_id = %s",
(acc_id,))
    result = cursor.fetchone()
    if result and result[0] >= amount:
        cursor.execute("UPDATE accounts SET balance = balance - %s WHERE
account_id = %s", (amount, acc_id))
        conn.commit()
        self.get_account_balance(acc_id)
    else:
        print(" Insufficient funds or account not found.")

def transfer(self, from_acc, to_acc, amount):
    cursor.execute("SELECT balance FROM accounts WHERE account_id = %s",
(from_acc,))
    result = cursor.fetchone()
    if result and result[0] >= amount:
        self.withdraw(from_acc, amount)
        self.deposit(to_acc, amount)
        print("Transfer successful.")
    else:
        print("Transfer failed. Check balance or account number.")

def get_account_details(self, acc_id):
    cursor.execute("""

```

```

        SELECT a.account_id, a.account_type, a.balance,
               c.first_name, c.last_name, c.email, c.phone_number, c.address
        FROM accounts a
        JOIN customers c ON a.customer_id = c.customer_id
        WHERE a.account_id = %s
        """ , (acc_id,))
    result = cursor.fetchone()
    if result:
        print(f"""
Account Number: {result[0]}
Account Type: {result[1]}
Balance: ₹{result[2]}
Customer Name: {result[3]} {result[4]}
Email: {result[5]}
Phone: {result[6]}
Address: {result[7]}
        """)
    else:
        print(" Account not found.")

def main():
    bank = Bank()
    while True:
        print("""
=== Banking System ===
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. Exit
        """)
        choice = input("Enter choice: ")

        if choice == '1':
            bank.create_account()
        elif choice == '2':

```

```
    acc = int(input("Enter Account Number: "))
    amt = float(input("Enter amount to deposit: "))
    bank.deposit(acc, amt)
elif choice == '3':
    acc = int(input("Enter Account Number: "))
    amt = float(input("Enter amount to withdraw: "))
    bank.withdraw(acc, amt)
elif choice == '4':
    acc = int(input("Enter Account Number: "))
    bank.get_account_balance(acc)
elif choice == '5':
    from_acc = int(input("From Account Number: "))
    to_acc = int(input("To Account Number: "))
    amt = float(input("Amount to transfer: "))
    bank.transfer(from_acc, to_acc, amt)
elif choice == '6':
    acc = int(input("Enter Account Number: "))
    bank.get_account_details(acc)
elif choice == '7':
    print("Exiting")
    break
else:
    print("Invalid choice. Try again.")

if __name__ == "__main__":
    main()
```

```
"C:\Users\ADMIN\PycharmProjects\Banking system\venv\Scripts\python.exe" "C:\Users\ADMIN\PycharmProjects\Banking system\main.py"

=== Banking System ===
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. Exit

Enter choice: 5
From Account Number: 2
To Account Number: 4
Amount to transfer: 500
Current Balance: ₹2500
Current Balance: ₹500
Transfer successful.

=== Banking System ===
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
```

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

Abstract Class:

An **abstract class** is a class that **cannot be instantiated** directly. It is used as a blueprint for other classes. It can contain:

- **Abstract methods** (methods without implementation)
- **Concrete methods** (normal methods)

Interface:

Python doesn't have a direct `interface` keyword like Java, but an abstract class **with only abstract methods** behaves like an interface

Static Variable

A **static variable** (also called class variable) is **shared across all instances** of the class. It is defined **outside of any instance method**, usually directly inside the class.

1. Create a 'Customer' class as mentioned above task.

```
class Customer:  
    def __init__(self, customer_id, first_name, last_name, email, phone_number, address):  
        self.customer_id = customer_id  
        self.first_name = first_name  
        self.last_name = last_name  
        self.email = email  
        self.phone_number = phone_number  
        self.address = address
```

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

```
class Account:  
    lastAccNo = 1000 # static variable  
  
    def __init__(self, account_type, balance, customer):  
        Account.lastAccNo += 1  
        self.account_number = Account.lastAccNo  
        self.account_type = account_type  
        self.balance = balance  
        self.customer = customer
```

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.

from bean.Account import Account

class SavingsAccount(Account):

```
def __init__(self, balance, customer, interest_rate=0.03):
    if balance < 500:
        raise ValueError("Savings account requires a minimum balance of 500.")
    super().__init__("Savings", balance, customer)
    self.interest_rate = interest_rate
```

from bean.Account import Account

class CurrentAccount(Account):

```
def __init__(self, balance, customer, overdraft_limit=1000):
    super().__init__("Current", balance, customer)
    self.overdraft_limit = overdraft_limit
```

```
def withdraw(self, amount):
```

```
    if self.balance + self.overdraft_limit >= amount:
        self.balance -= amount
        return self.balance
```

```
    else:
```

```
        raise ValueError("Withdrawal exceeds overdraft limit.")
```

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.

from abc import ABC, abstractmethod

class ICustomerServiceProvider(ABC):

def get_account_balance(self, account_number): pass

def deposit(self, account_number, amount): pass

def withdraw(self, account_number, amount): pass

def transfer(self, from_account_number, to_account_number, amount): pass

def get_account_details(self, account_number): pass

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.

from abc import ABC, abstractmethod

class IBankServiceProvider(ABC):

def create_account(self, customer, acc_type, balance): pass

def list_accounts(self): pass

def calculate_interest(self): pass

6. Create CustomerServiceProviderImpl class which implements ICustomerServiceProvider provide all implementation methods.

```
from service.ICustomerServiceProvider import ICustomerServiceProvider
```

```
class CustomerServiceProviderImpl(ICustomerServiceProvider):
```

```
    def __init__(self):
```

```
        self.conn = get_connection()
```

```
        self.cursor = self.conn.cursor()
```

```
    def get_account_balance(self, account_number):
```

```
        self.cursor.execute("SELECT balance FROM accounts WHERE acc_id = %s",  
(account_number,))
```

```
        result = self.cursor.fetchone()
```

```
        return result[0] if result else None
```

```
    def deposit(self, account_number, amount):
```

```
        self.cursor.execute("UPDATE accounts SET balance = balance + %s WHERE acc_id  
= %s", (amount, account_number))
```

```
        self.conn.commit()
```

```
        return self.get_account_balance(account_number)
```

```
    def withdraw(self, account_number, amount):
```

```
        balance = self.get_account_balance(account_number)
```

```
        if balance and balance >= amount:
```

```
            self.cursor.execute("UPDATE accounts SET balance = balance - %s WHERE  
acc_id = %s", (amount, account_number))
```

```
            self.conn.commit()
```

```
            return self.get_account_balance(account_number)
```

```
        else:
```

```
            raise ValueError("Insufficient funds.")
```

```
    def transfer(self, from_account_number, to_account_number, amount):
```

```
        self.withdraw(from_account_number, amount)
```

```
        self.deposit(to_account_number, amount)
```

```
    def get_account_details(self, account_number):
```

```
        self.cursor.execute("SELECT * FROM accounts WHERE acc_id = %s",
```

```
(account_number,))  
    return self.cursor.fetchone()
```

7. Create BankServiceProviderImpl class which inherits from CustomerServiceProviderImpl and implements IBankServiceProvider

- Attributes o accountList: Array of Accounts to store any account objects.

o branchName and branchAddress as String objects

```
from service.IBankServiceProvider import IBankServiceProvider  
from service.CustomerServiceProviderImpl import CustomerServiceProviderImpl
```

```
class BankServiceProviderImpl(CustomerServiceProviderImpl,  
IBankServiceProvider):
```

```
    def __init__(self, branch_name, branch_address):  
        super().__init__()  
        self.accountList = []  
        self.branchName = branch_name  
        self.branchAddress = branch_address  
  
    def create_account(self, customer, acc_type, balance):  
        # Insert customer and account into database  
        self.cursor.execute("INSERT INTO cust (first_name, last_name, email,  
phone_number, address) VALUES (%s, %s, %s, %s, %s)",  
                             (customer.first_name, customer.last_name, customer.email,  
customer.phone_number, customer.address))  
        self.conn.commit()  
        customer_id = self.cursor.lastrowid  
        self.cursor.execute("INSERT INTO acc (acc_type, balance, cust_id) VALUES  
(%s, %s, %s)",  
                             (acc_type, balance, customer_id))  
        self.conn.commit()  
  
    def list_accounts(self):  
        self.cursor.execute("SELECT * FROM acc")  
        return self.cursor.fetchall()
```

```

def calculate_interest(self):
    self.cursor.execute("SELECT acc_id, balance FROM acc WHERE acc_type =
'Savings'")
    for acc_id, balance in self.cursor.fetchall():
        interest = balance * 0.03
        self.cursor.execute("UPDATE acc SET balance = balance + %s WHERE acc_id
= %s", (interest, acc_id))
    self.conn.commit()

```

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.

```

from bean.Customer import Customer
from service.BankServiceProviderImpl import BankServiceProviderImpl

```

```

bank = BankServiceProviderImpl("HexaBank", "Chennai")

```

```

def main():
    while True:
print("\n--- Bank Menu ---")
        print("1. Create Account")
        print("2. Deposit")
        print("3. Withdraw")
        print("4. Get Balance")
        print("5. Transfer")
        print("6. Get Account Details")
        print("7. List Accounts")
        print("8. Exit")

```

```
choice = input("Enter choice: ")

if choice == '1':
    fname = input("First Name: ")
    lname = input("Last Name: ")
    email = input("Email: ")
    phone = input("Phone: ")
    address = input("Address: ")
    acc_type = input("Account Type (Savings/Current/ZeroBalance): ")
    balance = float(input("Initial Balance: "))

    customer = Customer(None, fname, lname, email, phone, address)
    bank.create_account(customer, acc_type, balance)
    print("Account created successfully!")

elif choice == '2':
    acc = int(input("Account Number: "))
    amt = float(input("Amount to deposit: "))
    print("New Balance:", bank.deposit(acc, amt))

elif choice == '3':
    acc = int(input("Account Number: "))
    amt = float(input("Amount to withdraw: "))
    print("New Balance:", bank.withdraw(acc, amt))

elif choice == '4':
    acc = int(input("Account Number: "))
    print("Balance:", bank.get_account_balance(acc))

elif choice == '5':
    from_acc = int(input("From Account: "))
    to_acc = int(input("To Account: "))
    amt = float(input("Amount: "))
    bank.transfer(from_acc, to_acc, amt)
    print("Transfer successful.")

elif choice == '6':
    acc = int(input("Account Number: "))
```

```

        print("Details:", bank.get_account_details(acc))

    elif choice == '7':
        for acc in bank.list_accounts():
            print(acc)

    elif choice == '8':
        print("Exiting...")
        break

    else:
        print("Invalid option.")

if __name__ == '__main__':
    main()

```

```

--- Bank Menu ---
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Exit
Enter choice: 1
First Name: Keerthi
Last Name: C
Email: keethi@gmail.com
Phone: 852147963
Address: usa
Account Type (Savings/Current/ZeroBalance): Saving
Initial Balance: 10

```

Task 12: Exception Handling

Exception handling: allows you to gracefully manage errors that occur during program execution, preventing crashes and ensuring smooth user experience. It involves using try, except, else, and finally blocks to catch and handle exceptions.

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.

```
class InsufficientFundException(Exception): def init(self, message="Insufficient funds in the account."): super().init(message)
```

2. InvalidAccountException throw this exception when user entered the invalid account number when tries to transfer amount, get account details classes.

```
class InvalidAccountException(Exception): def init(self, message="Account number is invalid."): super().init(message)
```

3. OverDraftLimitExceededException throw this exception when current account customer try to with draw amount from the current account.

```
class OverDraftLimitExceededException(Exception): def init(self, message="Overdraft limit exceeded."): super().init(message)
```

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.

```
from exception.InsufficientFundException import InsufficientFundException  
from exception.InvalidAccountException import InvalidAccountException  
from exception.OverDraftLimitExceededException import  
OverDraftLimitExceededException
```

```
def main():
```

```
    try:
```

```
        while True:
```

```
            print("\n=== Banking System ===")
```

```
            print("1. Create Account\n2. Deposit\n3. Withdraw\n4. Get Balance\n5.
```

```
Transfer\n6. Get Account Details\n7. Exit")
```

```
            choice = input("Enter your choice: ")
```

```
            if choice == "1":
```

```

        bank.create_account()
    elif choice == "2":
        acc_no = int(input("Enter account number: "))
        amount = float(input("Enter amount: "))
        print("Balance:", bank.deposit(acc_no, amount))
    elif choice == "3":
        acc_no = int(input("Enter account number: "))
        amount = float(input("Enter amount to withdraw: "))
        print("Balance:", bank.withdraw(acc_no, amount))
    elif choice == "4":
        acc_no = int(input("Enter account number: "))
        print("Balance:", bank.get_account_balance(acc_no))
    elif choice == "5":
        from_acc = int(input("From account: "))
        to_acc = int(input("To account: "))
        amt = float(input("Amount: "))
        bank.transfer(from_acc, to_acc, amt)
        print("Transfer successful.")
    elif choice == "6":
        acc_no = int(input("Enter account number: "))
        print(bank.get_account_details(acc_no))
    elif choice == "7":
        break
    else:
        print("Invalid choice.")

except (InsufficientFundException, InvalidAccountException,
OverDraftLimitExceededException) as e:
    print("Error:", e)
except Exception as e:
    print("An unexpected error occurred:", e)
if __name__ == "__main__":
    main()

```

InsufficientFundException:

When used?

- During **withdraw** or **transfer**

- If balance is not enough to perform the transaction

InvalidAccountException:

When used?

- When a user enters an **invalid account number**
- For **getAccountDetails**, **transfer**, etc

OverDraftLimitExceededException:

When used?

- In **CurrentAccount**, when the withdrawal amount exceeds the overdraft + balance

```
=== Banking System ===
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. Exit
Enter your choice: 4
Enter account number: 14
An unexpected error occurred: name 'bank' is not defined
```

Task 13: Collection

Collections:

refer to **data structures** that allow you to store and manage groups of related objects.

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 object to sort the accounts based on customer name when listAccounts() method called.

```
from exception.InsufficientFundException import InsufficientFundException
```

```
from exception.InvalidAccountException import InvalidAccountException
```

```
from exception.OverDraftLimitExceededException import  
OverDraftLimitExceededException
```

```
class HMBankList:
```

```
    def __init__(self):
```

```
        self.accounts = []
```

```
    def add_account(self, account):
```

```
        self.accounts.append(account)
```

```
    def list_accounts(self):
```

```
        for acc in self.accounts:
```

```
            print(acc)
```

```
    def withdraw(self, account_number, amount):
```

```
        account = self.find_account(account_number)
```

```
        if account is None:
```

```
            raise InvalidAccountException()
```

```
        if isinstance(account, CurrentAccount):
```

```
            if amount > (account.balance + account.overdraft_limit):
```

```
                raise OverDraftLimitExceededException()
```

```

        account.balance -= amount
    elif isinstance(account, SavingsAccount):
        if account.balance - amount < 500:
            raise InsufficientFundException("Minimum balance of ₹500 must be
maintained.")
        account.balance -= amount
    elif isinstance(account, ZeroBalanceAccount):
        if account.balance < amount:
            raise InsufficientFundException()
        account.balance -= amount

    return account.balance

def transfer(self, from_acc, to_acc, amount):
    sender = self.find_account(from_acc)
    receiver = self.find_account(to_acc)

    if sender is None or receiver is None:
        raise InvalidAccountException("One or both account numbers are invalid.")

    self.withdraw(from_acc, amount)
    self.deposit(to_acc, amount)

```

3. From the previous task change the HMBank attribute Accounts to HashMap of Accounts and perform the same operation.

```

import mysql.connector

from bean.customer import Customer

```

```

from bean.SavingsAccount import SavingsAccount

from bean.CurrentAccount import CurrentAccount

from bean.ZeroBalanceAccount import ZeroBalanceAccount


class HMBankMap:

    def __init__(self):

        self.accounts = {} # Dictionary: account_number -> account object

        self.conn = mysql.connector.connect(

            host="localhost",

            user="root",

            password="root@123", # Replace with your DB password

            database="bank" # Replace with your DB name

        )

        self.cursor = self.conn.cursor()


    def add_account(self, account):

        self.accounts[account.account_number] = account


    def create_account(self, customer, acc_type, balance, interest_rate=0.0,
overdraft_limit=0.0):

        # Save customer first

        self.cursor.execute("""

            INSERT INTO cust (first_name, last_name, email, phone_number, address)

            VALUES (%s, %s, %s, %s, %s)

            """, (customer.first_name, customer.last_name, customer.email,
customer.phone_number, customer.address))

```

```

self.conn.commit()

cust_id = self.cursor.lastrowid

if acc_type == "Savings":

    account = SavingsAccount(acc_type, balance, customer, interest_rate)

elif acc_type == "Current":

    account = CurrentAccount(acc_type, balance, customer, overdraft_limit)

elif acc_type == "ZeroBalance":

    account = ZeroBalanceAccount(acc_type, 0.0, customer)

else:

    raise ValueError("Invalid account type")

self.cursor.execute("""

    INSERT INTO acc (acc_type, balance, cust_id)

    VALUES (%s, %s, %s)

""", (account.account_type, account.balance, cust_id))

self.conn.commit()

account.account_number = self.cursor.lastrowid


self.add_account(account)

print(f"Account created successfully with Account Number:
{account.account_number}")


def deposit(self, acc_no, amount):

    account = self.get_account(acc_no)

    if account:

        account.balance += amount

```

```
        self.cursor.execute("UPDATE acc SET balance = %s WHERE acc_id = %s",
(account.balance, acc_no))
```

```
        self.conn.commit()
```

```
        print(f"Deposit successful. New Balance: ₹{account.balance}")
```

```
    else:
```

```
        print("Account not found.")
```

```
def withdraw(self, acc_no, amount):
```

```
    account = self.get_account(acc_no)
```

```
    if account:
```

```
        if hasattr(account, 'withdraw'):
```

```
            try:
```

```
                account.withdraw(amount)
```

```
                self.cursor.execute("UPDATE acc SET balance = %s WHERE acc_id = %s",
(account.balance, acc_no))
```

```
                self.conn.commit()
```

```
                print(f"Withdrawal successful. New Balance: ₹{account.balance}")
```

```
            except Exception as e:
```

```
                print(f"Error: {e}")
```

```
        else:
```

```
            print("Withdraw not supported for this account type.")
```

```
    else:
```

```
        print("Account not found.")
```

```
def transfer(self, from_acc_no, to_acc_no, amount):
```

```
    from_acc = self.get_account(from_acc_no)
```

```

to_acc = self.get_account(to_acc_no)

if from_acc and to_acc:

    try:

        if hasattr(from_acc, 'withdraw'):

            from_acc.withdraw(amount)

            to_acc.balance += amount

            self.cursor.execute("UPDATE acc SET balance = %s WHERE acc_id = %s",
(from_acc.balance, from_acc_no))

            self.cursor.execute("UPDATE acc SET balance = %s WHERE acc_id = %s",
(to_acc.balance, to_acc_no))

            self.conn.commit()

            print("Transfer successful.")

        else:

            print("Withdrawal not supported for source account.")

    except Exception as e:

        print(f"Error: {e}")

    else:

        print("One or both accounts not found.")


def get_account(self, acc_no):

    if acc_no in self.accounts:

        return self.accounts[acc_no]

    else:

        # Try loading from DB

        self.cursor.execute("SELECT a.acc_id, a.acc_type, a.balance, c.customer_id,
c.first_name, c.last_name, c.email, c.phone_number, c.address FROM acc a JOIN
cust c ON a.cust_id = c.customer_id WHERE a.acc_id = %s", (acc_no,))

```

```
result = self.cursor.fetchone()

if result:

    acc_id, acc_type, balance, cid, fname, lname, email, phone, addr = result

    cust = Customer(fname, lname, email, phone, addr)

    cust.customer_id = cid

    if acc_type == "Savings":

        account = SavingsAccount(acc_type, balance, cust)

    elif acc_type == "Current":

        account = CurrentAccount(acc_type, balance, cust)

    elif acc_type == "ZeroBalance":

        account = ZeroBalanceAccount(acc_type, balance, cust)

    else:

        return None

    account.account_number = acc_id

    self.accounts[acc_id] = account

    return account

return None
```

```
def get_account_details(self, acc_no):

    account = self.get_account(acc_no)

    if account:

        print(f"Account Number: {account.account_number}")

        print(f"Account Type: {account.account_type}")

        print(f"Balance: ₹{account.balance}")

        print(f"Customer: {account.customer.first_name} {account.customer.last_name}")
```


else:

print("Account not found.")

def list_accounts(self):

if not self.accounts:

print("No accounts available.")

for acc in self.accounts.values():

self.get_account_details(acc.account_number)

```
=== Banking System ===
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. Exit
Enter your choice: 4
Enter account number: 14
An unexpected error occurred: name 'bank' is not defined
```

Task 14: Database Connectivity.

Database connectivity means connecting your Python program to a database like **MySQL**, to **store, retrieve, update, or delete** data persistently.

In my project, i have used:

- **Database:** MySQL
- **Connector:** mysql-connector-python library
- **DB config file:** db_config.py to manage credentials

1. Create a 'Customer' class as mentioned above task.

class Customer:

def __init__(self, customer_id, first_name, last_name, email, phone_number, address):

```
self.customer_id = customer_id  
self.first_name = first_name  
self.last_name = last_name  
self.email = email  
self.phone_number = phone_number  
self.address = address
```

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

class Account:

```
lastAccNo = 1000
```

```
def __init__(self, account_type, balance, customer):
```

```
Account.lastAccNo += 1
```

```
self.account_number = Account.lastAccNo
```

```
self.account_type = account_type
```

```
self.balance = balance
```

```
self.customer = customer
```

3. Create a class 'TRANSACTION' that include following attributes

- Account
- Description
- Date and Time
- TransactionType(Withdraw, Deposit, Transfer)
- TransactionAmount

```
from datetime import datetime
```

```
class Transaction:
```

```
    def __init__(self, account, txn_type, amount, description=""):
```

```
        self.account = account
```

```
        self.txn_type = txn_type
```

```
        self.amount = amount
```

```
        self.description = description
```

```
        self.txn_time = datetime.now()
```

```
    def __str__(self):
```

```
        return f"{self.txn_type} of ₹{self.amount} on  
{self.txn_time.strftime('%Y-%m-%d %H:%M:%S')} - {self.description}"
```

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.

```
from bean.Account import Account
```

```
class SavingsAccount(Account):
```

```
    def __init__(self, balance, customer, interest_rate=0.03):
```

```
        if balance < 500:
```

```
            raise ValueError("Savings account requires a minimum balance of 500.")
```

```
        super().__init__("Savings", balance, customer)
```

```
        self.interest_rate = interest_rate
```

- CurrentAccount: A Current account that includes an additional attribute for overdraftLimit(credit limit).

```
from bean.Account import Account
```

```
class CurrentAccount(Account):
```

```
def __init__(self, balance, customer, overdraft_limit=1000):  
    super().__init__("Current", balance, customer)  
    self.overdraft_limit = overdraft_limit
```

```
def withdraw(self, amount):  
    if self.balance + self.overdraft_limit >= amount:  
        self.balance -= amount  
        return self.balance  
    else:  
        raise ValueError("Withdrawal exceeds overdraft limit.")
```

- ZeroBalanceAccount: ZeroBalanceAccount can be created with Zero balance.

```
from bean.Account import Account  
class ZeroBalanceAccount(Account):  
    def __init__(self, customer):  
        super().__init__("ZeroBalance", 0.0, customer)
```

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.
 - o A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.
 - o 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.
- `getTransactions(account_number: long, FromDate:Date, ToDate: Date)`: Should return the list of transaction between two dates.

from abc import ABC, abstractmethod

class ICustomerServiceProvider(ABC):

def get_account_balance(self, account_number): pass

def deposit(self, account_number, amount): pass

def withdraw(self, account_number, amount): pass

def transfer(self, from_account_number, to_account_number, amount): pass

def get_account_details(self, account_number): pass

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.

from abc import ABC, abstractmethod

class IBankServiceProvider(ABC):

def create_account(self, customer, acc_type, balance): pass

def list_accounts(self): pass

def calculate_interest(self): pass

7. Create CustomerServiceProviderImpl class which implements ICustomerServiceProvider provide all implementation methods. These methods do not interact with database directly.

```
from service.ICustomerServiceProvider import ICustomerServiceProvider  
from exception.InsufficientFundException import InsufficientFundException  
from exception.InvalidAccountException import InvalidAccountException  
from exception.OverDraftLimitExceededException import  
OverDraftLimitExceededException
```

```
class CustomerServiceProviderImpl(ICustomerServiceProvider):  
  
    def __init__(self):  
        self.conn = get_connection()  
        self.cursor = self.conn.cursor()  
  
    def get_account_balance(self, account_number):  
        self.cursor.execute("SELECT balance FROM accounts WHERE acc_id = %s",  
(account_number,))  
  
        result = self.cursor.fetchone()  
  
        return result[0] if result else None  
  
    def deposit(self, account_number, amount):  
        self.cursor.execute("UPDATE accounts SET balance = balance + %s WHERE acc_id  
= %s", (amount, account_number))  
  
        self.conn.commit()  
  
        return self.get_account_balance(account_number)
```

```

def withdraw(self, account_number, amount):

    balance = self.get_account_balance(account_number)

    if balance and balance >= amount:

        self.cursor.execute("UPDATE accounts SET balance = balance - %s WHERE
acc_id = %s", (amount, account_number))

        self.conn.commit()

        return self.get_account_balance(account_number)

    else:

        raise ValueError("Insufficient funds.")


def transfer(self, from_account_number, to_account_number, amount):

    self.withdraw(from_account_number, amount)

    self.deposit(to_account_number, amount)


def get_account_details(self, account_number):

    self.cursor.execute("SELECT * FROM accounts WHERE acc_id = %s",
(account_number,))

    return self.cursor.fetchone()

```

8. Create BankServiceProviderImpl class which inherits from CustomerServiceProviderImpl and implements IBankServiceProvider.

- Attributes

- o accountList: List of Accounts to store any account objects.
- o transactionList: List of Transaction to store transaction objects.
- o branchName and branchAddress as String objects

```

from service.CustomerServiceProviderImpl import CustomerServiceProviderImpl
from service.IBankServiceProvider import IBankServiceProvider

```

```
from bean.Transaction import Transaction

from bean.Account import Account

from exception.InsufficientFundException import InsufficientFundException

from exception.InvalidAccountException import InvalidAccountException

from exception.OverDraftLimitExceededException import
OverDraftLimitExceededException

class BankServiceProviderImpl(CustomerServiceProviderImpl,
IBankServiceProvider):

    def __init__(self, branchName, branchAddress):

        super().__init__()

        self.accountList = [] # List[Account]

        self.transactionList = [] # List[Transaction]

        self.branchName = branchName

        self.branchAddress = branchAddress


    def create_account(self, customer, accNo, accType, balance):

        # Logic for creating a new account (delegated to repository later)

        pass

    def listAccounts(self):

        return self.accountList

    def getAccountDetails(self, account_number):

        for account in self.accountList:

            if account.account_number == account_number:

                return account

        return None

    def calculateInterest(self):
```


for acc in self.accountList:

if hasattr(acc, 'interest_rate'):

acc.account_balance += acc.account_balance * acc.interest_rate / 100

9. Create IBankRepository 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 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.
 - o A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.
 - o 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 new 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.

from abc import ABC, abstractmethod

```

class IBankRepository(ABC):

    def createAccount(self, customer, accNo, accType, balance): pass

    def listAccounts(self): pass

    def calculateInterest(self): pass

    def getAccountBalance(self, account_number): pass

    def deposit(self, account_number, amount): pass

    def withdraw(self, account_number, amount): pass

    def transfer(self, from_account_number, to_account_number, amount): pass

    def getAccountDetails(self, account_number): pass

    def getTransactions(self, account_number, from_date, to_date): pass

```

10. Create BankRepositoryImpl class which implement the IBankRepository interface/abstract class and provide implementation of all methods and perform the database operations.

```

from service.IBankRepository import IBankRepository

from util.DBUtil import DBUtil

import mysql.connector

```

```

class BankRepositoryImpl(IBankRepository):

    def createAccount(self, customer, accNo, accType, balance):

        conn = DBUtil.getDBConn()

        cursor = conn.cursor()

        query = "INSERT INTO accounts (acc_no, acc_type, balance, customer_id) VALUES (%s, %s, %s, %s)"

        cursor.execute(query, (accNo, accType, balance, customer.customer_id))

        conn.commit()

        conn.close()

    def listAccounts(self):

```

```

    conn = DBUtil.getDBConn()

    cursor = conn.cursor(dictionary=True)

    cursor.execute("SELECT * FROM accounts")

    result = cursor.fetchall()

    conn.close()

    return result

def getAccountBalance(self, account_number):

    conn = DBUtil.getDBConn()

    cursor = conn.cursor()

    cursor.execute("SELECT balance FROM accounts WHERE acc_no = %s",
(account_number,))

    result = cursor.fetchone()

    conn.close()

    return result[0] if result else None

def deposit(self, account_number, amount):

    balance = self.getAccountBalance(account_number)

    new_balance = balance + amount

    conn = DBUtil.getDBConn()

    cursor = conn.cursor()

    cursor.execute("UPDATE accounts SET balance = %s WHERE acc_no = %s",
(new_balance, account_number))

    conn.commit()

    conn.close()

    return new_balance

def withdraw(self, account_number, amount):

```

```
balance = self.getAccountBalance(account_number)

if balance < amount:

    raise Exception("Insufficient Funds")

new_balance = balance - amount

conn = DBUtil.getDBConn()

cursor = conn.cursor()

cursor.execute("UPDATE accounts SET balance = %s WHERE acc_no = %s",
(new_balance, account_number))

conn.commit()

conn.close()

return new_balance

def transfer(self, from_account_number, to_account_number, amount):

    self.withdraw(from_account_number, amount)

    self.deposit(to_account_number, amount)


def getAccountDetails(self, account_number):

    conn = DBUtil.getDBConn()

    cursor = conn.cursor(dictionary=True)

    cursor.execute("SELECT * FROM accounts WHERE acc_no = %s",
(account_number,))

    result = cursor.fetchone()

    conn.close()

    return result

def getTransactions(self, account_number, from_date, to_date):

    conn = DBUtil.getDBConn()

    cursor = conn.cursor(dictionary=True)
```

```
query = "SELECT * FROM transactions WHERE acc_no = %s AND date_time  
BETWEEN %s AND %s"
```

```
cursor.execute(query, (account_number, from_date, to_date))
```

```
result = cursor.fetchall()
```

```
conn.close()
```

```
return result
```

```
def calculateInterest(self):
```

```
    Pass
```

11. Create DBUtil class and add the following method.

- static getDBConn():Connection Establish a connection to the database and return Connection reference

```
import mysql.connector
```

```
class DBUtil:
```

```
    def getDBConn():
```

```
        return mysql.connector.connect(
```

```
            host="localhost",
```

```
            user="root",
```

```
            password="root@123",
```

```
            database="bank"
```

```
        )
```

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

```
from service.BankServiceProviderImpl import BankServiceProviderImpl
```

```
from service.BankRepositoryImpl import BankRepositoryImpl
```

```
from bean.customer import Customer
```

```
from datetime import datetime
```

```
import traceback
```

```
def main():
```

```
    repo = BankRepositoryImpl()
```

```
    while True:
```

```
        print("\n==== HexaBank Menu =====")
```

```
        print("1. Create Account")
```

```
        print("2. Deposit")
```

```
        print("3. Withdraw")
```

```
        print("4. Get Balance")
```

```
        print("5. Transfer")
```

```
        print("6. Get Account Details")
```

```
        print("7. List All Accounts")
```

```
        print("8. Get Transactions")
```

```
        print("9. Exit")
```

```
    choice = input("Enter your choice: ")
```

```
    try:
```

```
        if choice == "1":
```

```
name = input("Enter customer name: ")
email = input("Enter email: ")
phone = input("Enter phone: ")
customer = Customer(name, email, phone)
acc_no = int(input("Enter account number: "))
acc_type = input("Enter account type (Savings/Current/ZeroBalance): ")
balance = float(input("Enter initial balance: "))
repo.createAccount(customer, acc_no, acc_type, balance)
print("Account created successfully!")

elif choice == "2":

    acc_no = int(input("Enter account number: "))
    amount = float(input("Enter deposit amount: "))
    new_balance = repo.deposit(acc_no, amount)
    print(f"Amount deposited successfully! New Balance: {new_balance}")

elif choice == "3":

    acc_no = int(input("Enter account number: "))
    amount = float(input("Enter withdrawal amount: "))
    new_balance = repo.withdraw(acc_no, amount)
    print(f"Amount withdrawn successfully! New Balance: {new_balance}")

elif choice == "4":

    acc_no = int(input("Enter account number: "))
    balance = repo.getAccountBalance(acc_no)
    print(f"Current Balance: {balance}")
```

elif choice == "5":

from_acc = int(input("Enter sender account number: "))

to_acc = int(input("Enter receiver account number: "))

amount = float(input("Enter transfer amount: "))

repo.transfer(from_acc, to_acc, amount)

print("Transfer completed successfully!")

elif choice == "6":

acc_no = int(input("Enter account number: "))

details = repo.getAccountDetails(acc_no)

if details:

print(details)

else:

print("Account not found")

elif choice == "7":

accounts = repo.listAccounts()

for acc in accounts:

print(acc)

elif choice == "8":

acc_no = int(input("Enter account number: "))

from_date = input("Enter From date (YYYY-MM-DD): ")

to_date = input("Enter To date (YYYY-MM-DD): ")

transactions = repo.getTransactions(acc_no, from_date, to_date)

for t in transactions:

print(t)


```
elif choice == "9":

    print("Thank you for using HexaBank!")

    break

else:

    print("Invalid choice. Please try again.")

except Exception as e:

    print("An error occurred:", e)

    traceback.print_exc()

main()
```

```
===== HexaBank Menu =====
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List All Accounts
8. Get Transactions
9. Exit
Enter your choice: 7
{'account_id': 1, 'account_type': 'savings', 'balance': 1000, 'customer_id': '3'}
{'account_id': 2, 'account_type': 'current ', 'balance': 3000, 'customer_id': '2'}
{'account_id': 3, 'account_type': 'savings', 'balance': 245, 'customer_id': '3'}
{'account_id': 4, 'account_type': 'zero_balance', 'balance': 0, 'customer_id': '4'}
{'account_id': 5, 'account_type': 'current', 'balance': 233, 'customer_id': '5'}
{'account_id': 6, 'account_type': 'savings', 'balance': 688, 'customer_id': '6'}
{'account_id': 7, 'account_type': 'current', 'balance': 900, 'customer_id': '3'}
{'account_id': 8, 'account_type': 'zero_balance', 'balance': 0, 'customer_id': '8'}
{'account_id': 9, 'account_type': 'savings', 'balance': 2000, 'customer_id': '9'}
{'account_id': 10, 'account_type': 'current', 'balance': 600, 'customer_id': '10'}
{'account_id': 11, 'account_type': 'Savings', 'balance': 40000, 'customer_id': None}
```

