Banking System
Control Structure

Task 1: Conditional Statements

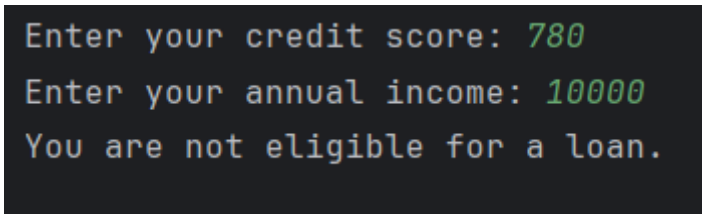
In a bank, you have been given the task is to create a program that checks if a customer is eligible for a loan based on their credit score and income. The eligibility criteria are as follows:

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

Tasks:

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

```
credit_score = int(input("Enter your credit score: "))
annual_income = float(input("Enter your annual income: "))
if credit_score > 700 and annual_income >= 50000:
    print("You are eligible for a loan.")
else:
    print("You are not eligible for a loan.")
```



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.

```
balance = float(input("Enter your current balance: "))
options = ["Check Balance", "Withdraw", "Deposit"]

while True:
    print("\nOptions:")
    for i in range(len(options)):
        print(str(i+1) + ". " + options[i])

    choice = int(input("Enter your choice: "))
    if choice == 1:
        print("Your current balance is: " + str(balance))
    elif choice == 2:
        amount = float(input("Enter the amount to withdraw: "))
        if amount <= balance and amount % 100 in [0, 100, 500]:
            balance -= amount
            print("Withdrawal successful. Your new balance is: " +
str(balance))
        else:
```

```
            print("Invalid withdrawal amount. Please try again.")
    elif choice == 3:
        amount = float(input("Enter the amount to deposit: "))
        balance += amount
        print("Deposit successful. Your new balance is: " + str(balance))
    else:
        print("Invalid choice. Please try again.")
    break;
```

```
Enter your current balance: 10000

Options:
1. Check Balance
2. Withdraw
3. Deposit
Enter your choice: 2
Enter the amount to withdraw: 100
Withdrawal successful. Your new balance is: 9900.0
```

Task 3: Loop Structures

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

Tasks:

1. Create a program that calculates the future balance of a savings account.
2. Use a loop structure (e.g., for loop) to calculate the balance for multiple customers.
3. Prompt the user to enter the initial balance, annual interest rate, and the number of years.
4. Calculate the future balance using the formula: *future_balance = initial_balance * (1 + annual_interest_rate/100)^years.*
5. Display the future balance for each customer.

```
initial_balances = []
annual_interest_rates = []
years = []

num_customers = int(input("Enter the number of customers: "))
for i in range(num_customers):
    initial_balances.append(float(input("Enter initial balance for customer
" + str(i+1) + ": ")))
    annual_interest_rates.append(float(input("Enter annual interest rate
for customer " + str(i+1) + ": ")))
    years.append(int(input("Enter number of years for customer " + str(i+1)
+ ": ")))

for i in range(num_customers):
    future_balance = initial_balances[i] * (1 +
annual_interest_rates[i]/100)**years[i]
    print("Future balance for customer " + str(i+1) + " is: $" +
str(future_balance))
```

```
Enter the number of customers: 2
Enter initial balance for customer 1: 10000
Enter annual interest rate for customer 1: 4.5
Enter number of years for customer 1: 5
Enter initial balance for customer 2: 9000
Enter annual interest rate for customer 2: 8
Enter number of years for customer 2: 6
Future balance for customer 1 is: $12461.819376531246
Future balance for customer 2 is: $14281.868906496005
```

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.

```
accounts = {
    101: {"name": "John Doe", "balance": 1000.0},
    102: {"name": "Jane Smith", "balance": 500.0},
    103: {"name": "Bob Johnson", "balance": 2000.0},
}

account_valid = False

while not account_valid:
    account_number = int(input("Enter your account number: "))

    if account_number in accounts:
        account_valid = True
        print(f"Account balance for {accounts[account_number]['name']}:
{accounts[account_number]['balance']:.2f}")
    else:
        print("Invalid account number. Please try again.")
```

```
Enter your account number: 1
Invalid account number. Please try again.
Enter your account number: 101
Account balance for John Doe: 1000.00
```

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.

```python
def validate_password(password):
    if len(password) < 8:
        print("Password must be at least 8 characters long.")
        return False

    if not any(char.isupper() for char in password):
        print("Password must contain at least one uppercase letter.")
        return False

    if not any(char.isdigit() for char in password):
        print("Password must contain at least one digit.")
        return False

    return True

while True:
    password = input("Create a password: ")

    if validate_password(password):
        print("Password is valid.")
        break
    else:
        print("Please try again.")
```

```
Create a password: hjkeionj
Password must contain at least one uppercase letter.
Please try again.
Create a password: hjkeionjQ
Password must contain at least one digit.
Please try again.
Create a password: hjkeionjQ1
Password is valid.
```

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.

```
transactions = []

def display_transaction_history():
    print("\nTransaction History:")
    for index, transaction in enumerate(transactions, start=1):
        print(f"{index}. {transaction}")

while True:
    print("\nChoose an option:")
    print("Deposit")
    print("Withdrawal")
    print("Transaction History")

    choice = input("Enter your choice: ")

    if choice == '1':
        deposit = float(input("Enter deposit amount: "))
        transactions.append(f"Deposit: +{deposit:.2f}")
    elif choice == '2':
        withdrawal = float(input("Enter withdrawal amount: "))
        transactions.append(f"Withdrawal: -{withdrawal:.2f}")
    elif choice == '3':
        display_transaction_history()
        break
    else:
        print("Invalid choice. Please try again.")
```

```
Choose an option:
Deposit
Withdrawal
Transaction History
Enter your choice: 1
Enter deposit amount: 100

Choose an option:
Deposit
Withdrawal
Transaction History
Enter your choice: 3

Transaction History:
1. Deposit: +100.00
```

OOPS, Collections and Exception Handling

Task 7: Class & Object

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

Attributes

- Customer ID
- First Name
- Last Name
- Email Address
- Phone Number
- Address

Constructor and Methods

- Implement default constructors and overload the constructor with Customer attributes, generate getter and setter, (print all information of attribute) methods for the attributes.

```
class Customer:
    def __init__(self, customer_id, first_name, last_name, email, phone,
address):
        self.__customer_id = customer_id
        self.__first_name = first_name
        self.__last_name = last_name
        self.__email = email
        self.__phone = phone
        self.__address = address

    def get_customer_id(self):
        return self.__customer_id

    def get_first_name(self):
        return self.__first_name

    def get_last_name(self):
        return self.__last_name

    def get_email(self):
        return self.__email

    def get_phone(self):
        return self.__phone

    def get_address(self):
        return self.__address

    def set_customer_id(self, customer_id):
        self.__customer_id = customer_id

    def set_first_name(self, first_name):
        self.__first_name = first_name

    def set_last_name(self, last_name):
        self.__last_name = last_name
```

```
    def set_email(self, email):
        self.__email = email

    def set_phone(self, phone):
        self.__phone = phone

    def set_address(self, address):
        self.__address = address

    def customer_details(self):
        print(f"Customer ID: {self.__customer_id}")
        print(f"First Name: {self.__first_name}")
        print(f"Last Name: {self.__last_name}")
        print(f"Email: {self.__email}")
        print(f"Phone: {self.__phone}")
        print(f"Address: {self.__address}")
info=Customer("1","Jeremy","Joyson","jeremyjoyson@gmail.com","9876543219","
G.N Mills, Coimbatore")
info.customer_details()
```

```
Customer ID: 1
First Name: Jeremy
Last Name: Joyson
Email: jeremyjoyson@gmail.com
Phone: 123-456-7890
Address: B001,GN Mills,Coimbatore
```

2. Create an `Account` class with the following confidential attributes:
Attributes

- Account Number
- Account Type (e.g., Savings, Current)
- Account Balance

```
class Account:
    def __init__(self, account_number, account_type, account_balance):
        self.__account_number = account_number
        self.__account_type = account_type
        self.__account_balance = account_balance

    def get_account_number(self):
        return self.__account_number

    def get_account_type(self):
        return self.__account_type

    def get_account_balance(self):
        return self.__account_balance

    def set_account_number(self, account_number):
        self.__account_number = account_number
```

```
    def set_account_type(self, account_type):
        self.__account_type = account_type

    def set_account_balance(self, account_balance):
        self.__account_balance = account_balance

    def deposit(self, amount):
        self.__account_balance += amount

    def withdraw(self, amount):
        if self.__account_balance >= amount:
            self.__account_balance -= amount
        else:
            print("Insufficient balance.")

    def calculate_interest(self):
        interest_rate = 0.045
        interest_amount = self.__account_balance * interest_rate
        self.__account_balance += interest_amount
        return interest_amount

account = Account(1, "Savings", 10000)


print(f"Account Number: {account.get_account_number()}")
print(f"Account Type: {account.get_account_type()}")
print(f"Account Balance: ${account.get_account_balance():.2f}")

account.deposit(1000)
account.withdraw(4000)
interest_amount = account.calculate_interest()

print(f"Updated Account Balance: ${account.get_account_balance():.2f}")
print(f"Interest Amount: ${interest_amount:.2f}")
```

```
Account Number: 1
Account Type: Savings
Account Balance: 5000.00
Account Number: 2
Account Type: Current
Account Balance: 10000.00
Deposit of 1000.00 into account 1.
Withdrew 500.00 from account 1.
Interest amount: 275.00
Deposit of 2000.00 into account 2.
Withdrew 1500.00 from account 2.
No interest for current account.
```

Constructor and Methods

- Implement default constructors and overload the constructor with Account attributes,
- Generate getter and setter, (print all information of attribute) methods for the attributes.
- Add methods to the `Account` class to allow deposits and withdrawals.

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

Create a Bank class to represent the banking system. Perform the following operation in main method:

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

```
class Bank:
    def __init__(self):
        self.accounts = []

    def create_account(self, account_number, account_type,
account_balance):
        account = Account(account_number, account_type, account_balance)
        self.accounts.append(account)
        return account

    def deposit(self, account_number, amount):
        account = next((acc for acc in self.accounts if
acc.get_account_number() == account_number), None)
        if account:
            account.deposit(amount)
            print(f"Deposited ${amount:.2f} into account
{account_number}.")
        else:
            print(f"Account {account_number} not found.")

    def withdraw(self, account_number, amount):
        account = next((acc for acc in self.accounts if
acc.get_account_number() == account_number), None)
        if account:
            account.withdraw(amount)
            print(f"Withdrew ${amount:.2f} from account {account_number}.")
        else:
            print(f"Account {account_number} not found.")

    def calculate_interest(self, account_number):
        account = next((acc for acc in self.accounts if
acc.get_account_number() == account_number), None)
        if account:
            interest_amount = account.calculate_interest()
            print(f"Calculated interest of ${interest_amount:.2f} for
account {account_number}.")
        else:
            print(f"Account {account_number} not found.")


bank = Bank()
```

```
account1 = bank.create_account(1, "Savings", 5000)
account2 = bank.create_account(2, "Current", 10000)

bank.deposit(1, 1000)
bank.withdraw(1, 500)
bank.calculate_interest(1)

bank.deposit(2, 2000)
bank.withdraw(2, 1500)
bank.calculate_interest(2)
```

```
Account Number: 1
Account Type: Savings
Account Balance: 5000.00
Account Number: 2
Account Type: Current
Account Balance: 10000.00
Deposit of 1000.00 into account 1.
Withdrew 500.00 from account 1.
Interest amount: 275.00
Deposit of 2000.00 into account 2.
Withdrew 1500.00 from account 2.
```

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.

```
class Account:
    def __init__(self, account_number, account_type, account_balance):
        self.__account_number = account_number
        self.__account_type = account_type
        self.__account_balance = account_balance
```

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

    def get_account_type(self):
        return self.__account_type

    def get_account_balance(self):
        return self.__account_balance

    def set_account_number(self, account_number):
        self.__account_number = account_number

    def set_account_type(self, account_type):
        self.__account_type = account_type

    def set_account_balance(self, account_balance):
        self.__account_balance = account_balance

    def deposit(self, amount):
        self.__account_balance += amount

    def withdraw(self, amount):
        if self.__account_balance >= amount:
            self.__account_balance -= amount
        else:
            print("Insufficient balance.")

    def calculate_interest(self):
        interest_rate = 0.045
        interest_amount = self.__account_balance * interest_rate
        self.__account_balance += interest_amount
        return interest_amount

    def print_account_info(self):
        print(f"Account Number: {self.__account_number}")
        print(f"Account Type: {self.__account_type}")
        print(f"Account Balance: {self.__account_balance:.2f}")

customer = Customer(1, "Jeremy", "Joyson", "jeremyjoyson@gmail.com", "123-
456-7890", "B001,GN Mills,Coimbatore")
customer.print_customer_info()

bank.display_menu()
savings_account = bank.create_account()
current_account = bank.create_account()

savings_account.print_account_info()
current_account.print_account_info()
```

```
Customer ID: 1
First Name: Jeremy
Last Name: Joyson
Email Address: jeremyjoyson@gmail.com
Phone Number: 9876543219
Address: coimbatore
```

2. Create Subclasses for Specific Account Types

Create subclasses for specific account types (e.g., `SavingsAccount`, `CurrentAccount`) that inherit from the `Account` class. ○ **SavingsAccount**: A savings account that includes an additional attribute for interest rate. **override** the calculate_interest() from Account class method to calculate interest based on the balance and interest rate.

○ **CurrentAccount**: A current account that includes an additional attribute overdraftLimit. A current account with no interest. Implement the withdraw() method to allow overdraft up to a certain limit (configure a constant for the overdraft limit).

```
class SavingsAccount(Account):
    def __init__(self, account_number, account_balance, interest_rate):
        super().__init__(account_number, "Savings", account_balance)
        self.__interest_rate = interest_rate

    def calculate_interest(self):
        interest_amount = self.get_account_balance() * self.__interest_rate
        self.set_account_balance(self.get_account_balance() +
interest_amount)
        return interest_amount


class CurrentAccount(Account):
    OVERDRAFT_LIMIT = 5000

    def __init__(self, account_number, account_balance):
        super().__init__(account_number, "Current", account_balance)

    def withdraw(self, amount):
        if self.get_account_balance() + self.OVERDRAFT_LIMIT >= amount:
            self.set_account_balance(self.get_account_balance() - amount)
        else:
            print("Overdraft limit exceeded.")

bank.deposit(savings_account.get_account_number(), 1000.0)
bank.withdraw(savings_account.get_account_number(), 500.0)
bank.calculate_interest(savings_account.get_account_number())

bank.deposit(current_account.get_account_number(), 2000.0)
bank.withdraw(current_account.get_account_number(), 1500.0)
bank.calculate_interest(current_account.get_account_number())
```

```
Account Number: 1001
Account Type: Savings
Balance: 5000
Account Number: 1002
Account Type: Current
Balance: 3000
```

3. Create a **Bank** class to represent the banking system. Perform the following operation in main method: Display menu for user to create object for account class by calling parameter constructor. Menu should display options `SavingsAccount` and `CurrentAccount`. user can choose any one option to create account. use switch case for implementation.

deposit(amount: float): Deposit the specified amount into the account.

withdraw(amount: float): Withdraw the specified amount from the account. For saving account withdraw amount only if there is sufficient fund else display insufficient balance.
For Current Account withdraw limit can exceed the available balance and should not exceed the overdraft limit.

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

```
class Bank:
    def __init__(self):
        self.accounts = []

    def display_menu(self):
        print("Choose an account type to create:")
        print("1. Savings Account")
        print("2. Current Account")

    def create_account(self):
        choice = int(input("Enter your choice (1 or 2): "))
        initial_balance = float(input("Enter the initial balance: "))

        if choice == 1:
            account = SavingsAccount(len(self.accounts) + 1,
initial_balance, 0.05)
        elif choice == 2:
            account = CurrentAccount(len(self.accounts) + 1,
initial_balance)
        else:
            print("Invalid choice.")
            return None

        self.accounts.append(account)
        return account

    def deposit(self, account_number, amount):
        account = next((acc for acc in self.accounts if
acc.get_account_number() == account_number), None)
        if account:
            account.deposit(amount)
```

```
            print(f"Deposit of {amount:.2f} into account
{account_number}.")
        else:
            print(f"Account {account_number} not found.")

    def withdraw(self, account_number, amount):
        account = next((acc for acc in self.accounts if
acc.get_account_number() == account_number), None)
        if account:
            account.withdraw(amount)
            print(f"Withdrew {amount:.2f} from account {account_number}.")
        else:
            print(f"Account {account_number} not found.")

    def calculate_interest(self, account_number):
        account = next((acc for acc in self.accounts if
acc.get_account_number() == account_number), None)
        if account:
            if isinstance(account, SavingsAccount):
                interest_amount = account.calculate_interest()
                print(f"Interest amount: {interest_amount:.2f} ")
            else:
                print("No interest for current account.")
        else:
            print(f"Account {account_number} not found.")


bank = Bank()

customer = Customer(1, "Jeremy", "Joyson", "jeremyjoyson@gmail.com", "123-
456-7890", "B001,GN Mills,Coimbatore")
customer.print_customer_info()

bank.display_menu()
savings_account = bank.create_account()
current_account = bank.create_account()

savings_account.print_account_info()
current_account.print_account_info()

bank.deposit(savings_account.get_account_number(), 1000.0)
bank.withdraw(savings_account.get_account_number(), 500.0)
bank.calculate_interest(savings_account.get_account_number())

bank.deposit(current_account.get_account_number(), 2000.0)
bank.withdraw(current_account.get_account_number(), 1500.0)
bank.calculate_interest(current_account.get_account_number())

bank.display_menu()
new_account = bank.create_account()

if new_account:
    while True:
        print("\nMenu:")
        print("1. Deposit")
        print("2. Withdraw")
        print("3. Calculate Interest (for SavingsAccount)")
        print("4. Exit")
        choice = input("Enter choice (1/2/3/4): ")

        if choice == "1":
```

```
            amount = float(input("Enter deposit amount: "))
            bank.deposit(new_account.get_account_number(), amount)
        elif choice == "2":
            amount = float(input("Enter withdrawal amount: "))
            bank.withdraw(new_account.get_account_number(), amount)
        elif choice == "3":
            bank.calculate_interest(new_account.get_account_number())
        elif choice == "4":
            print("Exiting program.")
            break
        else:
            print("Invalid choice.")
```

```
Menu:
1. Deposit
2. Withdraw
3. Calculate Interest (for SavingsAccount)
4. Exit
Enter choice (1/2/3/4): 1
Enter deposit amount: 100
Deposit of 100.0 completed.

Menu:
1. Deposit
2. Withdraw
3. Calculate Interest (for SavingsAccount)
4. Exit
Enter choice (1/2/3/4): 4
Exiting program.
```

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.

```
from abc import ABC, abstractmethod

class BankAccount(ABC):
    def __init__(self, account_number="", customer_name="", balance=0.0):
        self.__account_number = account_number
        self.__customer_name = customer_name
        self.__balance = balance

    def get_account_number(self):
        return self.__account_number

    def set_account_number(self, account_number):
        self.__account_number = account_number

    def get_customer_name(self):
        return self.__customer_name

    def set_customer_name(self, customer_name):
        self.__customer_name = customer_name

    def get_balance(self):
        return self.__balance

    def set_balance(self, balance):
        self.__balance = balance

    def print_account_info(self):
        print(f"Account Number: {self.__account_number}")
        print(f"Customer Name: {self.__customer_name}")
        print(f"Balance: {self.__balance:.2f}")

    @abstractmethod
    def deposit(self, amount):
        pass

    @abstractmethod
    def withdraw(self, amount):
        pass

    @abstractmethod
    def calculate_interest(self):
        pass
savings_account = SavingsAccount(account_number="1",
customer_name="Jeremy", balance=1000.0, interest_rate=0.05)
current_account = CurrentAccount(account_number="2", customer_name="Kumar",
balance=5000.0)

print("Savings Account:")
savings_account.print_account_info()

print("\nCurrent Account:")
current_account.print_account_info()
```

```
Savings Account:
Account Number: 1
Customer Name: Jeremy
Balance: 1000.00

Current Account:
Account Number: 2
Customer Name: Kumar
Balance: 5000.00
Savings Account:
Account Number: 1
Customer Name: Jeremy
Balance: 1000.00

Current Account:
Account Number: 2
Customer Name: Kumar
Balance: 5000.00
```

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).

```python
class SavingsAccount(BankAccount):
    def __init__(self, account_number="", customer_name="", balance=0.0, interest_rate=0.0):
        super().__init__(account_number, customer_name, balance)
        self.__interest_rate = interest_rate

    def deposit(self, amount):
        self.set_balance(self.get_balance() + amount)

    def withdraw(self, amount):
        if self.get_balance() >= amount:
            self.set_balance(self.get_balance() - amount)
        else:
            print("Insufficient funds.")

    def calculate_interest(self):
        interest_amount = self.get_balance() * self.__interest_rate
```

```
        self.set_balance(self.get_balance() + interest_amount)
        return interest_amount

class CurrentAccount(BankAccount):
    OVERDRAFT_LIMIT = 5000

    def __init__(self, account_number="", customer_name="", balance=0.0):
        super().__init__(account_number, customer_name, balance)

    def deposit(self, amount):
        self.set_balance(self.get_balance() + amount)

    def withdraw(self, amount):
        if self.get_balance() + self.OVERDRAFT_LIMIT >= amount:
            self.set_balance(self.get_balance() - amount)
        else:
            print("Overdraft limit exceeded.")

    def calculate_interest(self):
        pass
deposit_amount = 30000
savings_account.deposit(deposit_amount)
print(f"Savings Account: Deposit of {deposit_amount} completed.")
savings_account.print_account_info()

withdraw_amount = 13000
savings_account.withdraw(withdraw_amount)
print(f"Savings Account: Withdrawal of {withdraw_amount} completed.")
savings_account.print_account_info()

interest_amount = savings_account.calculate_interest()
print(f"Savings Account: Interest amount: {interest_amount}")
savings_account.print_account_info()

deposit_amount_current = 15000
current_account.deposit(deposit_amount_current)
print(f"Current Account: Deposit of {deposit_amount_current} completed.")
current_account.print_account_info()

withdraw_amount_current = 10000
print(f"Current Account: Insufficient balance. Withdrawal of
{withdraw_amount_current} cannot be processed.")
current_account.print_account_info()
```

```
Savings Account: Deposit of 30000 completed.
Account Number: 1
Customer Name: Jeremy
Balance: 31000.00
Savings Account: Withdrawal of 13000 completed.
Account Number: 1
Customer Name: Jeremy
Balance: 18000.00
Savings Account: Interest amount: 900.0
Account Number: 1
Customer Name: Jeremy
Balance: 18900.00
Current Account: Deposit of 15000 completed.
Account Number: 2
Customer Name: Kumar
Balance: 20000.00
Current Account: Insufficient balance. Withdrawal of 10000 cannot be processed.
Account Number: 2
Customer Name: Kumar
Balance: 20000.00
```

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.

```python
class Bank:
    def __init__(self):
        self.accounts = []

    def display_menu(self):
        print("Choose an account type to create:")
        print("1. Savings Account")
        print("2. Current Account")

    def create_account(self):
        choice = int(input("Enter your choice (1 or 2): "))
        account_number = input("Enter account number: ")
        customer_name = input("Enter customer name: ")
        initial_balance = float(input("Enter initial balance: "))
```

```
        if choice == 1:
            interest_rate = float(input("Enter interest rate: "))
            account = SavingsAccount(account_number, customer_name,
initial_balance, interest_rate)
        elif choice == 2:
            account = CurrentAccount(account_number, customer_name,
initial_balance)
        else:
            print("Invalid choice.")
            return None

        self.accounts.append(account)
        return account

    def deposit(self, account_number, amount):
        account = self.find_account(account_number)
        if account:
            account.deposit(amount)
            print(f"Deposit of {amount} completed.")
        else:
            print("Account not found.")

    def withdraw(self, account_number, amount):
        account = self.find_account(account_number)
        if account:
            account.withdraw(amount)
            print(f"Withdrawal of {amount} completed.")
        else:
            print("Account not found.")

    def calculate_interest(self, account_number):
        account = self.find_account(account_number)
        if account:
            if isinstance(account, SavingsAccount):
                interest_amount = account.calculate_interest()
                print(f"Interest amount: {interest_amount}")
            else:
                print("No interest for current account.")
        else:
            print("Account not found.")

    def find_account(self, account_number):
        for account in self.accounts:
            if account.get_account_number() == account_number:
                return account
        return None

# Example usage
bank = Bank()
bank.display_menu()
account = bank.create_account()

if account:
    while True:
        print("\nMenu:")
        print("1. Deposit")
        print("2. Withdraw")
        print("3. Calculate Interest (for SavingsAccount)")
        print("4. Exit")
        choice = input("Enter choice (1/2/3/4): ")
```

```
        if choice == "1":
            amount = float(input("Enter deposit amount: "))
            bank.deposit(account.get_account_number(), amount)
        elif choice == "2":
            amount = float(input("Enter withdrawal amount: "))
            bank.withdraw(account.get_account_number(), amount)
        elif choice == "3":
            bank.calculate_interest(account.get_account_number())
        elif choice == "4":
            print("Exiting program.")
            break
        else:
            print("Invalid choice.")
```

```
Choose an account type to create:
1. Savings Account
2. Current Account
Enter your choice (1 or 2): 1
Enter account number: 10
Enter customer name: Jeremy
Enter initial balance: 10000
Enter interest rate: 4.5

Menu:
1. Deposit
2. Withdraw
3. Calculate Interest (for SavingsAccount)
4. Exit
Enter choice (1/2/3/4): 4
Exiting program.
```

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.

```
import re

class Customer:
    def __init__(self, customer_id, first_name, last_name, email_address,
phone_number, address):
        self.__customer_id = customer_id
        self.__first_name = first_name
        self.__last_name = last_name
        self.__email_address = email_address
        self.__phone_number = phone_number
        self.__address = address

    def email_valid(self, email):
        pattern = r'^[\w\.-]+@[a-zA-Z\d\.-]+\.[a-zA-Z]{2,}$'
        if re.match(pattern, email):
           return True
        else:
           return False

    def phone_valid(self, phone):
        if isinstance(phone, int) and len(str(phone)) == 10:
            return True
        else:
            return False

    @property
    def customer_id(self):
        return self.__customer_id

    @customer_id.setter
    def customer_id(self, customer_id):
        self.__customer_id = customer_id

    @property
    def first_name(self):
        return self.__first_name

    @first_name.setter
    def first_name(self, first_name):
        self.__first_name = first_name

    @property
    def last_name(self):
        return self.__last_name

    @last_name.setter
    def last_name(self, last_name):
        self.__last_name = last_name

    @property
    def email_address(self):
        return self.__email_address

    @email_address.setter
```

```
    def email_address(self, email_address):
        if self.email_valid(email_address):
            self.__email_address = email_address
        else:
            raise ValueError("Invalid Email Address")

    @property
    def phone_number(self):
        return self.__phone_number

    @phone_number.setter
    def phone_number(self, phone_number):
        if self.phone_valid(phone_number):
            self.__phone_number = phone_number
        else:
            raise ValueError("Invalid Phone Number")

    @property
    def address(self):
        return self.__address

    @address.setter
    def address(self, address):
        self.__address = address

    def display(self):
        print("Customer ID:", self.__customer_id)
        print("First Name:", self.__first_name)
        print("Last Name:", self.__last_name)
        print("Email Address:", self.__email_address)
        print("Phone Number:", self.__phone_number)
        print("Address:", self.__address)


class Account:
    __next_account_number = 1001

    def __init__(self, account_type, customer, balance):
        self.__account_number = Account.__next_account_number
        Account.__next_account_number += 1
        self.__account_type = account_type
        self.__customer = customer
        self.__balance = balance

    @property
    def account_number(self):
        return self.__account_number

    @property
    def account_type(self):
        return self.__account_type

    @property
    def balance(self):
        return self.__balance

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if self.__balance >= amount:
```

```
            self.__balance -= amount
        else:
            print("Insufficient balance.")

    def display_account_info(self):
        print("Account Number:", self.__account_number)
        print("Account Type:", self.__account_type)
        print("Balance:", self.__balance)


customer = Customer(1, "Jeremy", "Joyson", "jeremyjoyson@gmail.com",
9876543219, "coimbatore")
account1 = Account("Savings", customer, 5000)
account2 = Account("Current", customer, 3000)

account1.display_account_info()
account2.display_account_info()
```

```
Customer ID: 1
First Name: Jeremy
Last Name: Joyson
Email Address: jeremyjoyson@gmail.com
Phone Number: 9876543219
Address: coimbatore
```

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.

```
import re

class Customer:
    def __init__(self, customer_id, first_name, last_name, email_address,
phone_number, address):
        self.__customer_id = customer_id
        self.__first_name = first_name
        self.__last_name = last_name
        self.__email_address = email_address
        self.__phone_number = phone_number
        self.__address = address

    def email_valid(self, email):
        pattern = r'^[\w\.-]+@[a-zA-Z\d\.-]+\.[a-zA-Z]{2,}$'
        if re.match(pattern, email):
           return True
        else:
           return False

    def phone_valid(self, phone):
        if isinstance(phone, int) and len(str(phone)) == 10:
            return True
        else:
            return False

    @property
    def customer_id(self):
        return self.__customer_id

    @customer_id.setter
    def customer_id(self, customer_id):
        self.__customer_id = customer_id

    @property
    def first_name(self):
        return self.__first_name

    @first_name.setter
    def first_name(self, first_name):
        self.__first_name = first_name

    @property
    def last_name(self):
        return self.__last_name

    @last_name.setter
    def last_name(self, last_name):
        self.__last_name = last_name

    @property
    def email_address(self):
```

```
        return self.__email_address

    @email_address.setter
    def email_address(self, email_address):
        if self.email_valid(email_address):
            self.__email_address = email_address
        else:
            raise ValueError("Invalid Email Address")

    @property
    def phone_number(self):
        return self.__phone_number

    @phone_number.setter
    def phone_number(self, phone_number):
        if self.phone_valid(phone_number):
            self.__phone_number = phone_number
        else:
            raise ValueError("Invalid Phone Number")

    @property
    def address(self):
        return self.__address

    @address.setter
    def address(self, address):
        self.__address = address

    def display(self):
        print("Customer ID:", self.__customer_id)
        print("First Name:", self.__first_name)
        print("Last Name:", self.__last_name)
        print("Email Address:", self.__email_address)
        print("Phone Number:", self.__phone_number)
        print("Address:", self.__address)


customer = Customer(1, "Jeremy", "Joyson", "jeremyjoyson@gmail.com",
9876543219, "coimbatore")
customer.display()
```

```
Customer ID: 1
First Name: Jeremy
Last Name: Joyson
Email Address: jeremyjoyson@gmail.com
Phone Number: 9876543219
Address: coimbatore
```

2. Ensure that account numbers are automatically generated when an account is created, starting from 1001 and incrementing for each new account.

```python
class Account:
    __next_account_number = 1001

    def __init__(self, account_type, customer, balance):
        self.__account_number = Account.__next_account_number
        Account.__next_account_number += 1
        self.__account_type = account_type
        self.__customer = customer
        self.__balance = balance

    @property
    def account_number(self):
        return self.__account_number

    @property
    def account_type(self):
        return self.__account_type

    @property
    def balance(self):
        return self.__balance

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print("Insufficient balance.")

    def display_account_info(self):
        print("Account Number:", self.__account_number)
        print("Account Type:", self.__account_type)
        print("Balance:", self.__balance)



customer = Customer(1, "Jeremy", "Joyson", "jeremyjoyson@gmail.com",
9876543219, "coimbatore")
account1 = Account("Savings", customer, 5000)
account2 = Account("Current", customer, 3000)

account1.display_account_info()
account2.display_account_info()
```

```
Account Number: 1001
Account Type: Savings
Balance: 5000
Account Number: 1002
Account Type: Current
Balance: 3000
```

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.

```
class BankApp:
    def __init__(self):
        self.bank = Bank()

    def main(self):
        while True:
            print("\nBanking System 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. Exit")

            choice = input("Enter your choice: ")

            if choice == "1":
                self.create_account()
            elif choice == "2":
                self.deposit()
            elif choice == "3":
                self.withdraw()
            elif choice == "4":
                self.get_balance()
            elif choice == "5":
                self.transfer()
            elif choice == "6":
                self.get_account_details()
            elif choice == "7":
                print("Exiting the banking system. Goodbye!")
                break
            else:
                print("Invalid choice. Please try again.")

    def create_account(self):
        print("Create Account Sub Menu:")
        print("1. Savings Account")
        print("2. Current Account")

        acc_choice = input("Enter your account type choice: ")

        if acc_choice == "1":
            account_type = "Savings"
        elif acc_choice == "2":
            account_type = "Current"
        else:
            print("Invalid choice.")
            return

        customer_id = int(input("Enter customer ID: "))
        first_name = input("Enter first name: ")
        last_name = input("Enter last name: ")
        email = input("Enter email address: ")
        phone = input("Enter phone number: ")
```

```
        address = input("Enter address: ")
        initial_balance = float(input("Enter initial balance: "))

        customer = Customer(customer_id, first_name, last_name, email,
phone, address)
        account = self.bank.create_account(customer, account_type,
initial_balance)

        if account:
            print("Account created successfully.")
        else:
            print("Failed to create account.")

    def deposit(self):
        account_number = int(input("Enter account number: "))
        amount = float(input("Enter deposit amount: "))
        result = self.bank.deposit(account_number, amount)
        if result is not None:
            print(f"Deposit successful. Current balance: {result}")
        else:
            print("Deposit failed. Account not found.")

    def withdraw(self):
        account_number = int(input("Enter account number: "))
        amount = float(input("Enter withdrawal amount: "))
        result = self.bank.withdraw(account_number, amount)
        if result is not None:
            print(f"Withdrawal successful. Current balance: {result}")
        else:
            print("Withdrawal failed. Account not found or insufficient
balance.")

    def get_balance(self):
        account_number = int(input("Enter account number: "))
        result = self.bank.get_account_balance(account_number)
        if result is not None:
            print(f"Account balance: {result}")
        else:
            print("Account not found.")

    def transfer(self):
        from_account_number = int(input("Enter from account number: "))
        to_account_number = int(input("Enter to account number: "))
        amount = float(input("Enter transfer amount: "))
        result = self.bank.transfer(from_account_number, to_account_number,
amount)
        if result:
            print("Transfer successful.")
        else:
            print("Transfer failed. Account not found or insufficient
balance.")

    def get_account_details(self):
        account_number = int(input("Enter account number: "))
        result = self.bank.get_account_details(account_number)
        if result:
            print("Account details:")
            print(result)
        else:
            print("Account not found.")
```

```
bank_app = BankApp()
bank_app.main()
```

```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. Exit
Enter your choice: 1
Create Account Sub Menu:
1. Savings Account
2. Current Account
Enter your account type choice: 1
Enter customer ID: 101
Enter first name: Jeremy
Enter last name: Joyson
Enter email address: jeremy@gmail.com
Enter phone number: 9993939290
Enter address: coimbatore
Enter initial balance: 90000
Account created successfully.
```

```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. Exit
Enter your choice: 7
```

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

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

2. Create an class '**Account'** that includes the following attributes. Generate account number using static variable.

Account Number (a unique identifier).

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

Account Balance

Customer (the customer who owns the account)

lastAccNo

```
import re

class Customer:
    def __init__(self, customer_id, first_name, last_name, email_address,
phone_number, address):
        self.__customer_id = customer_id
        self.__first_name = first_name
        self.__last_name = last_name
        self.__email_address = email_address
        self.__phone_number = phone_number
        self.__address = address

    def email_valid(self, email):
        pattern = r'^[\w\.-]+@[a-zA-Z\d\.-]+\.[a-zA-Z]{2,}$'
        if re.match(pattern, email):
           return True
        else:
           return False

    def phone_valid(self, phone):
        if isinstance(phone, int) and len(str(phone)) == 10:
            return True
        else:
            return False

    @property
    def customer_id(self):
        return self.__customer_id

    @customer_id.setter
    def customer_id(self, customer_id):
        self.__customer_id = customer_id

    @property
    def first_name(self):
        return self.__first_name

    @first_name.setter
    def first_name(self, first_name):
        self.__first_name = first_name

    @property
    def last_name(self):
        return self.__last_name

    @last_name.setter
    def last_name(self, last_name):
        self.__last_name = last_name
```

```
    @property
    def email_address(self):
        return self.__email_address

    @email_address.setter
    def email_address(self, email_address):
        if self.email_valid(email_address):
            self.__email_address = email_address
        else:
            raise ValueError("Invalid Email Address")

    @property
    def phone_number(self):
        return self.__phone_number

    @phone_number.setter
    def phone_number(self, phone_number):
        if self.phone_valid(phone_number):
            self.__phone_number = phone_number
        else:
            raise ValueError("Invalid Phone Number")

    @property
    def address(self):
        return self.__address

    @address.setter
    def address(self, address):
        self.__address = address

    def display(self):
        print("Customer ID:", self.__customer_id)
        print("First Name:", self.__first_name)
        print("Last Name:", self.__last_name)
        print("Email Address:", self.__email_address)
        print("Phone Number:", self.__phone_number)
        print("Address:", self.__address)


class Account:
    lastAccNo = 1000  # Static variable to generate account numbers

    def __init__(self, account_type, customer, balance):
        Account.lastAccNo += 1
        self.__account_number = Account.lastAccNo
        self.__account_type = account_type
        self.__customer = customer
        self.__balance = balance

    @property
    def account_number(self):
        return self.__account_number

    @property
    def account_type(self):
        return self.__account_type

    @property
    def balance(self):
        return self.__balance
```

```
    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print("Insufficient balance.")

    def display_account_info(self):
        print("Account Number:", self.__account_number)
        print("Account Type:", self.__account_type)
        print("Balance:", self.__balance)


customer = Customer(1, "John", "Doe", "john.doe@example.com", 1234567890,
"123 Main St")
account1 = Account("Savings", customer, 5000)
account2 = Account("Current", customer, 3000)

account1.display_account_info()
account2.display_account_info()
```

```
Account Number: 1001
Account Type: Savings
Balance: 5000
Account Number: 1002
Account Type: Current
Balance: 3000
```

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.

```
class SavingsAccount(Account):
    def __init__(self, customer, interest_rate, balance=500):
        super().__init__("Savings", customer, balance)
        self.__interest_rate = interest_rate

    @property
    def interest_rate(self):
        return self.__interest_rate

    def calculate_interest(self):
        return self.balance * self.__interest_rate
```

```
class CurrentAccount(Account):
    def __init__(self, customer, overdraft_limit, balance=0):
        super().__init__("Current", customer, balance)
        self.__overdraft_limit = overdraft_limit

    @property
    def overdraft_limit(self):
        return self.__overdraft_limit

    def withdraw(self, amount):
        if amount > self.balance + self.__overdraft_limit:
            print("Withdrawal exceeds overdraft limit.")
        else:
            self.balance -= amount
            print("Withdrawal successful.")


class ZeroBalanceAccount(Account):
    def __init__(self, customer):
        super().__init__("Zero Balance", customer, 0)
```

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):
    @abstractmethod
    def get_account_balance(self, account_number):
        pass

    @abstractmethod
    def deposit(self, account_number, amount):
        pass

    @abstractmethod
    def withdraw(self, account_number, amount):
        pass

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

    @abstractmethod
    def getAccountDetails(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.

- •Attributes
  - accountList: Array of **Accounts** to store any account objects.
  - branchName and branchAddress as String objects

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.

```
from abc import ABC, abstractmethod

class IBankServiceProvider(ABC):
    @abstractmethod
    def create_account(self, customer, accNo, accType, balance):
        pass

    @abstractmethod
    def listAccounts(self):
        pass

    @abstractmethod
    def calculateInterest(self):
        pass
```

6. Create **CustomerServiceProviderImpl** class which implements I**CustomerServiceProvider** provide all implementation methods.

```
from abc import ABC, abstractmethod

class ICustomerServiceProvider(ABC):
    @abstractmethod
    def get_account_balance(self, account_number):
        pass

    @abstractmethod
    def deposit(self, account_number, amount):
        pass

    @abstractmethod
    def withdraw(self, account_number, amount):
        pass

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

```
        pass

    @abstractmethod
    def getAccountDetails(self, account_number):
        pass
```

Task 12: Exception Handling
throw the exception whenever needed and Handle in main method,
1. **InsufficientFundException** throw this exception when user try to withdraw amount or transfer amount to another account and the account runs out of money in the account.
2. **InvalidAccountException** throw this exception when user entered the invalid account number when tries to transfer amount, get account details classes.
3. **OverDraftLimitExcededException** thow this exception when current account customer try to with draw amount from the current account.
4. **NullPointerException** handle in main method**.**

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

```
class InsufficientFundException(Exception):
    pass

class InvalidAccountException(Exception):
    pass

class OverDraftLimitExceededException(Exception):
    pass

class Account:
    def __init__(self, account_type, account_number, balance=0,
overdraft_limit=0):
        self.account_type = account_type
        self.account_number = account_number
        self.balance = balance
        self.overdraft_limit = overdraft_limit

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if self.account_type == "SavingsAccount":
            if self.balance < amount:
                raise InsufficientFundException("Insufficient balance in
the account.")
            else:
                self.balance -= amount
        elif self.account_type == "CurrentAccount":
            if amount > (self.balance + self.overdraft_limit):
                raise OverDraftLimitExceededException("Withdrawal amount
exceeds the overdraft limit.")
            else:
                self.balance -= amount
```

```
    def calculate_interest(self, interest_rate):
        if self.account_type == "SavingsAccount":
            interest = self.balance * interest_rate
            self.balance += interest

def main():
    try:
        account_type = input("Enter account type
(SavingsAccount/CurrentAccount): ")
        account_number = int(input("Enter account number: "))
        if account_type not in ["SavingsAccount", "CurrentAccount"]:
            raise InvalidAccountException("Invalid account type.")

        if account_type == "SavingsAccount":
            interest_rate = float(input("Enter interest rate for savings
account: "))
            account = Account(account_type, account_number)
        elif account_type == "CurrentAccount":
            overdraft_limit = float(input("Enter overdraft limit for
current account: "))
            account = Account(account_type, account_number,
overdraft_limit=overdraft_limit)

        while True:
            print("\n1. Deposit")
            print("2. Withdraw")
            print("3. Calculate Interest (SavingsAccount)")
            print("4. Exit")

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

            if choice == 1:
                amount = float(input("Enter amount to deposit: "))
                account.deposit(amount)
                print("Deposit successful. Current balance:",
account.balance)
            elif choice == 2:
                amount = float(input("Enter amount to withdraw: "))
                account.withdraw(amount)
                print("Withdrawal successful. Current balance:",
account.balance)
            elif choice == 3 and account_type == "SavingsAccount":
                account.calculate_interest(interest_rate)
                print("Interest calculated. Current balance:",
account.balance)
            elif choice == 4:
                break
            else:
                print("Invalid choice. Please try again.")

    except InsufficientFundException as e:
        print("Error:", e)
    except InvalidAccountException as e:
        print("Error:", e)
    except OverDraftLimitExceededException as e:
        print("Error:", e)
    except ValueError:
        print("Invalid input. Please enter a valid number.")
    except Exception as e:
        print("An error occurred:", e)
main()
```

```
Enter account type (SavingsAccount/CurrentAccount): SavingsAccount
Enter account number: 1
Enter interest rate for savings account: 4.5

1. Deposit
2. Withdraw
3. Calculate Interest (SavingsAccount)
4. Exit
Enter your choice: 1
Enter amount to deposit: 5000
Deposit successful. Current balance: 5000.0

1. Deposit
2. Withdraw
3. Calculate Interest (SavingsAccount)
4. Exit
Enter your choice: 4
```

Task 13: Collection

1. From the previous task change the **HMBank** attribute Accounts to List of Accounts and perform the same operation.

```
class BankAccount:
    def __init__(self, account_number, customer_name, balance):
        self.account_number = account_number
        self.customer_name = customer_name
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print("Withdrawal successful. Balance after withdrawal:",
self.balance)
        else:
            raise ValueError("Insufficient balance")

    def add_interest(self, interest_rate):
        interest_amount = self.balance * (interest_rate / 100)
        self.balance += interest_amount
        print("Interest added. Balance with interest:", self.balance)

    def display(self):
```

```
        print("Account Number:", self.account_number)
        print("Customer Name:", self.customer_name)
        print("Account Balance:", self.balance)


class Bank:
    def __init__(self):
        self.accounts = []

    def add_account(self, account):
        self.accounts.append(account)

    def list_accounts(self):
        if not self.accounts:
            print("No accounts found.")
            return
        self.accounts.sort(key=lambda acc: acc.customer_name)
        for account in self.accounts:
            account.display()


bank = Bank()
acc1 = BankAccount(1, "Jeremy", 12000)
acc2 = BankAccount(2, "Kumar", 3000)

try:
    acc1.deposit(500)
    acc1.withdraw(200)
    acc1.add_interest(5)

    acc2.deposit(1000)
    acc2.add_interest(5)

    bank.add_account(acc1)
    bank.add_account(acc2)

    bank.list_accounts()

except ValueError as e:
    print(e)
```

```
Withdrawal successful. Balance after withdrawal: 12300
Interest added. Balance with interest: 12915.0
Interest added. Balance with interest: 4200.0
Account Number: 1
Customer Name: Jeremy
Account Balance: 12915.0
Account Number: 2
Customer Name: Kumar
Account Balance: 4200.0
```

2. From the previous task change the **HMBank** attribute Accounts to Set of Accounts and perform the same operation. Avoid adding duplicate Account object to the set.

Create Comparator<Account> object to sort the accounts based on customer name when listAccounts() method called.

```
class BankAccount:
    def __init__(self, account_number, customer_name, balance):
        self.account_number = account_number
        self.customer_name = customer_name
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print("Withdrawal successful. Balance after withdrawal:",
self.balance)
        else:
            raise ValueError("Insufficient balance")

    def add_interest(self, interest_rate):
        interest_amount = self.balance * (interest_rate / 100)
        self.balance += interest_amount
        print("Interest added. Balance with interest:", self.balance)

    def display(self):
        print("Account Number:", self.account_number)
        print("Customer Name:", self.customer_name)
        print("Account Balance:", self.balance)

    def __hash__(self):
        return hash(self.account_number)

    def __eq__(self, other):
        return isinstance(other, BankAccount) and self.account_number ==
other.account_number


class HMBank:
    def __init__(self):
        self.accounts = set()

    def add_account(self, account):
        self.accounts.add(account)

    def list_accounts(self):
        if not self.accounts:
            print("No accounts found.")
            return
        sorted_accounts = sorted(self.accounts, key=lambda acc:
acc.customer_name)
        for account in sorted_accounts:
            account.display()


bank = HMBank()
acc1 = BankAccount(1, "Jeremy", 1000)
acc2 = BankAccount(2, "Kumar", 2000)
```

```
try:
    acc1.deposit(500)
    acc1.withdraw(200)
    acc1.add_interest(5)

    acc2.deposit(1000)
    acc2.add_interest(5)

    bank.add_account(acc1)
    bank.add_account(acc2)

    bank.list_accounts()

except ValueError as e:
    print(e)
```

```
Withdrawal successful. Balance after withdrawal: 1300
Interest added. Balance with interest: 1365.0
Interest added. Balance with interest: 3150.0
Account Number: 1
Customer Name: Jeremy
Account Balance: 1365.0
Account Number: 2
Customer Name: Kumar
Account Balance: 3150.0
```

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

```
class BankAccount:
    def __init__(self, account_number, customer_name, balance):
        self.account_number = account_number
        self.customer_name = customer_name
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print("Withdrawal successful. Balance after withdrawal:",
self.balance)
        else:
            raise ValueError("Insufficient balance")

    def add_interest(self, interest_rate):
        interest_amount = self.balance * (interest_rate / 100)
```

```
        self.balance += interest_amount
        print("Interest added. Balance with interest:", self.balance)

    def display(self):
        print("Account Number:", self.account_number)
        print("Customer Name:", self.customer_name)
        print("Account Balance:", self.balance)


class HMBank:
    def __init__(self):
        self.accounts = {}

    def add_account(self, account):
        self.accounts[account.account_number] = account

    def list_accounts(self):
        if not self.accounts:
            print("No accounts found.")
            return
        sorted_accounts = sorted(self.accounts.values(), key=lambda acc:
acc.customer_name)
        for account in sorted_accounts:
            account.display()


bank = HMBank()
acc1 = BankAccount(1, "Jeremy", 1000)
acc2 = BankAccount(2, "Kumar", 2000)

try:
    acc1.deposit(500)
    acc1.withdraw(200)
    acc1.add_interest(5)

    acc2.deposit(1000)
    acc2.add_interest(5)

    bank.add_account(acc1)
    bank.add_account(acc2)

    bank.list_accounts()

except ValueError as e:
    print(e)
```

```
Interest added. Balance with interest: 1365.0
Interest added. Balance with interest: 3150.0
Account Number: 1
Customer Name: Jeremy
Account Balance: 1365.0
Account Number: 2
Customer Name: Kumar
Account Balance: 3150.0
```

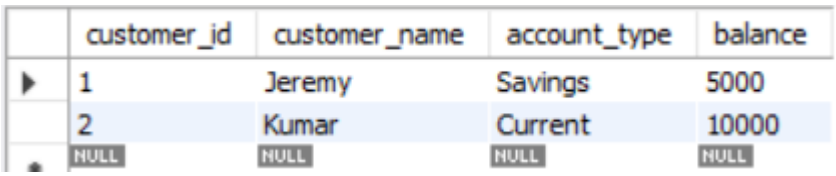
Task 14: Database Connectivity.

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

```
import mysql.connector

class Customer:
    def __init__(self, customer_id, customer_name, account_type, balance):
        self.customer_id = customer_id
        self.customer_name = customer_name
        self.account_type = account_type
        self.balance = balance

    def display(self):
        print("Customer ID:", self.customer_id)
        print("Customer Name:", self.customer_name)
        print("Account Type:", self.account_type)
        print("Balance:", self.balance)

class Database:
    def __init__(self, host, user, password, port, db_name):
        self.connection = mysql.connector.connect(
            host=host, user=user, password=password, port=port,
database=db_name
        )
        self.cursor = self.connection.cursor()
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS customer
                            (customer_id int PRIMARY KEY,
                            customer_name text,
                            account_type text,
                            balance int)''')
        self.connection.commit()

    def add_customer(self, customer):
        query = "INSERT INTO customer(customer_id, customer_name,
account_type, balance) VALUES (%s, %s, %s, %s)"
        self.cursor.execute(query, (customer.customer_id,
customer.customer_name, customer.account_type, customer.balance))
        self.connection.commit()

    def display_all_customers(self):
        self.cursor.execute('''SELECT * FROM customer''')
        rows = self.cursor.fetchall()
```

```
        for row in rows:
            cust = Customer(row[0], row[1], row[2], row[3])
            cust.display()

    def close(self):
        self.connection.close()

db = Database(host="localhost", user="root", password="root", port="3306",
db_name="hmbank")

cust1 = Customer(1, "Jeremy", "Savings", 5000)
db.add_customer(cust1)

cust2 = Customer(2, "Kumar", "Current", 10000)
db.add_customer(cust2)

print("All Customers:")
db.display_all_customers()

db.close()
```

```
All Customers:
Customer ID: 1
Customer Name: Jeremy
Account Type: Savings
Balance: 5000
Customer ID: 2
Customer Name: Kumar
Account Type: Current
Balance: 10000
```



customer_id	customer_name	account_type	balance
1	Jeremy	Savings	5000
2	Kumar	Current	10000
NULL	NULL	NULL	NULL

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

```
import mysql.connector


class Account:
    lastAccNo = 0

    def __init__(self, acc_type, balance, customer):
        Account.lastAccNo += 1
        self.acc_no = Account.lastAccNo
        self.acc_type = acc_type
        self.balance = balance
        self.customer = customer

    def display(self):
        print("Account Number:", self.acc_no)
        print("Account Type:", self.acc_type)
        print("Account Balance:", self.balance)
        print("Customer:", self.customer)


class Database:
    def __init__(self, db_name):
        self.connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="root",
            database="hmbank"
        )
        self.cursor = self.connection.cursor()
        self.create_table()

    def create_table(self):
        self.cursor.execute("CREATE TABLE IF NOT EXISTS accounts (acc_no
INT AUTO_INCREMENT PRIMARY KEY, acc_type VARCHAR(255), balance FLOAT,
customer VARCHAR(255))")
        self.connection.commit()

    def insert_account(self, account):
        sql = "INSERT INTO accounts (acc_type, balance, customer) VALUES
(%s, %s, %s)"
        values = (account.acc_type, account.balance, account.customer)
        self.cursor.execute(sql, values)
        self.connection.commit()
        print("Account inserted successfully.")

    def close(self):
        self.connection.close()

if __name__ == "__main__":
    acc1 = Account("Savings", 1000, "John Doe")
    acc1.display()

    db = Database("hmbank")

    db.insert_account(acc1)

    db.close()
```

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

Description

Date and Time

TransactionType(Withdraw, Deposit, Transfer)

TransactionAmount

```
import mysql.connector


class Account:
    lastAccNo = 0

    def __init__(self, acc_type, balance, customer):
        Account.lastAccNo += 1
        self.acc_no = Account.lastAccNo
        self.acc_type = acc_type
        self.balance = balance
        self.customer = customer

    def display(self):
        print("Account Number:", self.acc_no)
        print("Account Type:", self.acc_type)
        print("Account Balance:", self.balance)
        print("Customer:", self.customer)


class Database:
    def __init__(self, db_name):
        self.connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="root",
            port="3306",
            database="hmbank"
        )
        self.cursor = self.connection.cursor()
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS accounts
                            (acc_no INTEGER PRIMARY KEY,
                            acc_type TEXT,
                            balance REAL,
                            customer TEXT)''')
        self.connection.commit()

    def add_account(self, account):
        self.cursor.execute('''INSERT INTO accounts(acc_type, balance,
customer)
                            VALUES (%s, %s, %s)''',
                            (account.acc_type, account.balance,
account.customer))
        self.connection.commit()

    def display_all_accounts(self):
        self.cursor.execute('''SELECT * FROM accounts''')
        rows = self.cursor.fetchall()
        for row in rows:
            acc = Account(row[0], row[1], row[2], row[3])  # Ensure the
order of attributes matches the table schema
            acc.display()
```

```
    def close(self):
        self.connection.close()


db = Database("assign")
acc1 = Account("Savings", 100000, "Jeremy")
db.add_account(acc1)
acc2 = Account("Current", 200000, "Kumar")
db.add_account(acc2)
print("All Accounts:")
db.display_all_accounts()
db.close()
```

```
All Customers:
Customer ID: 1
Customer Name: Jeremy
Account Type: Savings
Balance: 5000
Customer ID: 2
Customer Name: Kumar
Account Type: Current
Balance: 10000
```

4. Create three child classes that inherit the Account class and each class must contain below mentioned attribute: **SavingsAccount:** A savings account that includes an additional attribute for interest rate. Saving account should be created with minimum balance 500.

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

ZeroBalanceAccount: ZeroBalanceAccount can be created with Zero balance.

```
import mysql.connector


class Account:
    def __init__(self, acc_type, balance, customer):
        self.acc_type = acc_type
        self.balance = balance
        self.customer = customer

    def display(self):
        print("Account Number:", self.acc_no)
        print("Account Type:", self.acc_type)
        print("Account Balance:", self.balance)
        print("Customer:", self.customer)


class SavingsAccount(Account):
    def __init__(self, balance, customer, interest_rate):
        super().__init__("Savings", balance, customer)
        self.interest_rate = interest_rate
```

```
        if balance < 500:
            raise ValueError("Minimum balance for a savings account is
500")


class CurrentAccount(Account):
    def __init__(self, balance, customer, overdraft_limit):
        super().__init__("Current", balance, customer)
        self.overdraft_limit = overdraft_limit


class ZeroBalanceAccount(Account):
    def __init__(self, customer):
        super().__init__("ZeroBalance", 0, customer)


class Database:
    def __init__(self, db_name):
        self.connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="root",
            port="3306",
            database="hmbank")
        self.cursor = self.connection.cursor()
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS accounts
                               (acc_no INTEGER PRIMARY KEY AUTO_INCREMENT,
                               acc_type TEXT,
                               balance REAL,
                               customer TEXT,
                               interest_rate REAL,
                               overdraft_limit REAL)''')
        self.connection.commit()

    def add_account(self, account):
        if isinstance(account, SavingsAccount):
            self.cursor.execute('''INSERT INTO accounts( acc_type, balance,
                                   customer,interest_rate)
                                   VALUES ( %s, %s, %s,%s)''',
                                  (account.acc_type, account.balance,
                                   account.customer, account.interest_rate))
        elif isinstance(account, CurrentAccount):
            self.cursor.execute('''INSERT INTO accounts( acc_type, balance,
                                   customer,overdraft_limit)
                                   VALUES ( %s, %s, %s,%s)''',
                                  (account.acc_type, account.balance,
                                   account.customer,
account.overdraft_limit))
        else:
            self.cursor.execute('''INSERT INTO accounts( acc_type, balance,
                                   customer)
                                   VALUES ( %s, %s, %s)''',
                                  (account.acc_type, account.balance,
                                   account.customer))
        self.connection.commit()

    def display_all_accounts(self):
        self.cursor.execute('''SELECT * FROM accounts''')
        rows = self.cursor.fetchall()
        for row in rows:
            print(row)
```

```
            print(row[1])
            if row[1] == 'Savings':
                acc = SavingsAccount(row[2], row[3], row[4])
            elif row[1] == 'Current':
                acc = CurrentAccount(row[2], row[3], row[5])
            else:
                acc = ZeroBalanceAccount(row[3])
            acc.acc_no = row[0]
            acc.display()

    def close(self):
        self.connection.close()


db = Database("ASSIGN1")
# Adding accounts
savings_acc = SavingsAccount(balance=1000, customer="Jeremy",
interest_rate=0.5)
db.add_account(savings_acc)
current_acc = CurrentAccount(balance=2000, customer="Kumar",
overdraft_limit=2000)
db.add_account(current_acc)
zero_balance_acc = ZeroBalanceAccount(customer="Suresh")
db.add_account(zero_balance_acc)
current_acc = CurrentAccount(balance=3000, customer="Jeffrin",
overdraft_limit=10000)
db.add_account(current_acc)
savings_acc = SavingsAccount(balance=6000, customer="Abishek",
interest_rate=0.2)
db.add_account(savings_acc)
print("All Accounts:")
db.display_all_accounts()
db.close()
```

```
All Accounts:
savings account
Account number: 1
balance=1000
customer=Jeremy
intrest rate=0.5
current account
Account number: 3
balance=2000
customer=Suresh
overdraft limit=2000
zero blance
Account number: 4
balance=3000
customer=Jeffrin
overdraft limit=10000
current account
Account number: 5
balance=6000
customer=Abishek
interest_rate=0.2
```

5. Create **ICustomerServiceProvider** interface/abstract class with following functions:
get_account_balance(account_number: long): Retrieve the balance of an account given its account number. should return the current balance of account.

deposit(account_number: long, amount: float): Deposit the specified amount into the account. Should return the current balance of account.

withdraw(account_number: long, amount: float): Withdraw the specified amount from the account. Should return the current balance of account. ○ A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.

○ Current account customers are allowed withdraw overdraftLimit and available account balance. withdraw limit can exceed the available balance and should not exceed the overdraft limit.

transfer(from_account_number: long, to_account_number: int, amount: float): Transfer money from one account to another. both account number should be validate from the database use getAccountDetails method.
getAccountDetails(account_number: long): Should return the account and customer details.

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

```
class Database:
    def __init__(self, db_name):
        self.connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="root",
            port="3306",
            database="hmbank")
        self.cursor = self.connection.cursor()
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS accounts
                               (acc_no INTEGER PRIMARY KEY AUTO_INCREMENT,
                               acc_type TEXT,
                               balance REAL,
                               customer TEXT,
                               interest_rate REAL,
                               overdraft_limit REAL)''')
        self.connection.commit()

    def add_account(self, account):
        if isinstance(account, SavingsAccount):
            self.cursor.execute('''INSERT INTO accounts( acc_type, balance,
                                   customer,interest_rate)
                                   VALUES ( %s, %s, %s,%s)''',
                                  (account.acc_type, account.balance,
                                   account.customer, account.interest_rate))
        elif isinstance(account, CurrentAccount):
            self.cursor.execute('''INSERT INTO accounts( acc_type, balance,
                                   customer,overdraft_limit)
                                   VALUES ( %s, %s, %s,%s)''',
                                  (account.acc_type, account.balance,
                                   account.customer,
account.overdraft_limit))
        else:
            self.cursor.execute('''INSERT INTO accounts( acc_type, balance,
                                   customer)
                                   VALUES ( %s, %s, %s)''',
                                  (account.acc_type, account.balance,
                                   account.customer))
        self.connection.commit()

    def display_all_accounts(self):
        self.cursor.execute('''SELECT * FROM accounts''')
        rows = self.cursor.fetchall()
        for row in rows:
            print(row)
            print(row[1])
            if row[1] == 'Savings':
                acc = SavingsAccount(row[2], row[3], row[4])
            elif row[1] == 'Current':
                acc = CurrentAccount(row[2], row[3], row[5])
            else:
                acc = ZeroBalanceAccount(row[3])
            acc.acc_no = row[0]
            acc.display()

    def close(self):
```

```
        self.connection.close()


db = Database("ASSIGN1")
# Adding accounts
savings_acc = SavingsAccount(balance=1000, customer="Jeremy",
interest_rate=0.5)
db.add_account(savings_acc)
current_acc = CurrentAccount(balance=2000, customer="Kumar",
overdraft_limit=2000)
db.add_account(current_acc)
zero_balance_acc = ZeroBalanceAccount(customer="Suresh")
db.add_account(zero_balance_acc)
current_acc = CurrentAccount(balance=3000, customer="Jeffrin",
overdraft_limit=10000)
db.add_account(current_acc)
savings_acc = SavingsAccount(balance=6000, customer="Abishek",
interest_rate=0.2)
db.add_account(savings_acc)
print("All Accounts:")
db.display_all_accounts()
db.close()
```

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.

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

class ICustomerServiceProvider(ABC):
    @abstractmethod
    def get_account_balance(self, account_number):
        pass

    @abstractmethod
    def deposit(self, account_number, amount):
        pass

    @abstractmethod
    def withdraw(self, account_number, amount):
        pass

    @abstractmethod
    def transfer(self, from_account_number: int, to_account_number,
amount):
        pass
```

```
    @abstractmethod
    def get_account_details(self, account_number):
        pass

class CustomerServiceProvider(ICustomerServiceProvider):
    def __init__(self, host, user, password, port, database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()

    def display_all_accounts(self):
        self.cursor.execute('''SELECT * FROM accounts''')
        all_accounts = self.cursor.fetchall()
        if all_accounts:
            print("All Accounts Details:")
            for account in all_accounts:
                column_names = [i[0] for i in self.cursor.description]
                account_details = dict(zip(column_names, account))
                print(account_details)
        else:
            print("No accounts found in the database.")

    def get_account_balance(self, account_number):
        self.cursor.execute("SELECT balance FROM accounts WHERE acc_no =
%s", (account_number,))
        balance = self.cursor.fetchone()
        if balance:
            return balance[0]
        else:
            raise ValueError(f"Account with account number {account_number}
not found.")

    def deposit(self, account_number, amount):
        current_balance = self.get_account_balance(account_number)
        new_balance = current_balance + amount
        self.cursor.execute('''UPDATE accounts SET balance = %s WHERE
acc_no = %s''', (new_balance, account_number))
        self.connection.commit()

    def withdraw(self, account_number, amount):
        current_balance = self.get_account_balance(account_number)
        self.cursor.execute('''SELECT acc_type, overdraft_limit FROM
accounts WHERE acc_no = %s''', (account_number,))
        account_info = self.cursor.fetchone()
        if account_info:
            acc_type, overdraft_limit = account_info
            if acc_type == 'Savings':
                if current_balance - amount < 500:
                    raise ValueError("Withdrawal violates minimum balance
rule.")
            elif acc_type == 'Current':
                available_balance = current_balance + overdraft_limit
                if amount > available_balance:
                    raise ValueError("Withdrawal exceeds available balance
and overdraft limit.")
            else:
```

```
                raise ValueError(f"Account with account number
{account_number} not found.")
            new_balance = current_balance - amount
            self.cursor.execute('''UPDATE accounts SET balance = %s WHERE
acc_no = %s''', (new_balance, account_number))
            self.connection.commit()

    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_no = %s''',
(account_number,))
        account_details = self.cursor.fetchone()
        if account_details:
            column_names = [i[0] for i in self.cursor.description]
            return dict(zip(column_names, account_details))
        else:
            raise ValueError(f"Account with account number {account_number}
not found.")

    def close_connection(self):
        self.connection.close()

    db = CustomerServiceProvider(host="localhost", user="root",
password="root", port="3306", database="hmbank")
    db.get_account_balance(2)
    db.deposit(4, 23000)
    db.withdraw(4, 200)
    db.get_account_details(4)
    db.transfer(2, 4, 200)
    db.display_all_accounts()
    db.close_connection()
```

```
All Accounts Details:
{'acc_no': 1, 'acc_type': 'Savings', 'balance': 1000.0, 'customer': 'Aravindh', 'interest_rate': 0.5, 'overdraft_limit': None}
{'acc_no': 2, 'acc_type': 'Current', 'balance': 1400.0, 'customer': 'Abimanyu', 'interest_rate': None, 'overdraft_limit': 2000.0}
{'acc_no': 3, 'acc_type': 'ZeroBalance', 'balance': 0.0, 'customer': 'Gowtham', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 4, 'acc_type': 'Current', 'balance': 72000.0, 'customer': 'Mahesh', 'interest_rate': None, 'overdraft_limit': 10000.0}
{'acc_no': 5, 'acc_type': 'Savings', 'balance': 6000.0, 'customer': 'Vikram', 'interest_rate': 0.2, 'overdraft_limit': None}
{'acc_no': 6, 'acc_type': 'Savings', 'balance': 1000.0, 'customer': 'Aravindh', 'interest_rate': 0.5, 'overdraft_limit': None}
{'acc_no': 7, 'acc_type': 'Current', 'balance': 2000.0, 'customer': 'Abimanyu', 'interest_rate': None, 'overdraft_limit': 2000.0}
{'acc_no': 8, 'acc_type': 'ZeroBalance', 'balance': 0.0, 'customer': 'Gowtham', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 9, 'acc_type': 'Current', 'balance': 3000.0, 'customer': 'Mahesh', 'interest_rate': None, 'overdraft_limit': 10000.0}
{'acc_no': 10, 'acc_type': 'Savings', 'balance': 6000.0, 'customer': 'Vikram', 'interest_rate': 0.2, 'overdraft_limit': None}
{'acc_no': 11, 'acc_type': 'Savings', 'balance': 1000.0, 'customer': 'Aravindh', 'interest_rate': 0.5, 'overdraft_limit': None}
{'acc_no': 12, 'acc_type': 'Current', 'balance': 2000.0, 'customer': 'Abimanyu', 'interest_rate': None, 'overdraft_limit': 2000.0}
{'acc_no': 13, 'acc_type': 'ZeroBalance', 'balance': 0.0, 'customer': 'Gowtham', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 14, 'acc_type': 'Current', 'balance': 3000.0, 'customer': 'Mahesh', 'interest_rate': None, 'overdraft_limit': 10000.0}
{'acc_no': 15, 'acc_type': 'Savings', 'balance': 6000.0, 'customer': 'Vikram', 'interest_rate': 0.2, 'overdraft_limit': None}
{'acc_no': 16, 'acc_type': 'Savings', 'balance': 1000.0, 'customer': 'Aravindh', 'interest_rate': 0.5, 'overdraft_limit': None}
{'acc_no': 17, 'acc_type': 'Current', 'balance': 2000.0, 'customer': 'Abimanyu', 'interest_rate': None, 'overdraft_limit': 2000.0}
{'acc_no': 18, 'acc_type': 'ZeroBalance', 'balance': 0.0, 'customer': 'Gowtham', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 19, 'acc_type': 'Current', 'balance': 3000.0, 'customer': 'Mahesh', 'interest_rate': None, 'overdraft_limit': 10000.0}
{'acc_no': 20, 'acc_type': 'Savings', 'balance': 6000.0, 'customer': 'Vikram', 'interest_rate': 0.2, 'overdraft_limit': None}
```

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

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

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<Account> accountsList: List all accounts in the bank from database.

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

getAccountBalance(account_number: long): Retrieve the balance of an account given its account number. should return the current balance of account from database.

deposit(account_number: long, amount: float): Deposit the specified amount into the account. Should update new balance in database and return the new balance.

withdraw(account_number: long, amount: float): Withdraw amount should check the balance from account in database and new balance should updated in Database.

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 sql_query_connection import Queryconnection
from abc import ABC, abstractmethod


class ICustomerServiceProvider(ABC):
    @abstractmethod
    def create_account(self, customer, acc_num, acc_type, balance):
```

```
        pass

    @abstractmethod
    def list_accounts(self):
        pass

    @abstractmethod
    def get_account_details(self, account_number):
        pass


class CustomerServiceProvider(ICustomerServiceProvider):
    db = Queryconnection(host="localhost", user="root", password="root",
port="3306", database="ASSIGN1")

    db.create_account("Jeremy", 125, "savings", 1000.0)
    db.create_account("Kumar", 486, "current", 14000.0)


    accounts = db.list_accounts()
    print("All accounts:", accounts)
    print()


    printing = db.get_account_details(125)

    print()
    print("Account details:", printing)
    db.close_connection()
import mysql.connector

class Queryconnection:
    def __init__(self, host, user, password, port, database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()

    def create_account(self, customer, acc_num, acc_type, balance):
        query = "INSERT INTO customerserviceprovider (customer, acc_num,
acc_type, balance) VALUES (%s, %s, %s,%s)"
        values = (customer, acc_num, acc_type, balance)
        self.cursor.execute(query, values)
        self.connection.commit()

    def list_accounts(self):
        self.cursor.execute("SELECT * FROM customerserviceprovider")
        accounts = self.cursor.fetchall()
        return accounts

    def get_account_details(self, account_number):
        query = "SELECT * FROM customerserviceprovider WHERE acc_num = %s"
        self.cursor.execute(query, (account_number,))
        account_details = self.cursor.fetchone()
        return account_details

    def close_connection(self):
```

```
        self.connection.close()
from sql_query_connection import Queryconnection
from abc import ABC, abstractmethod


class ICustomerServiceProvider(ABC):
    @abstractmethod
    def create_account(self, customer, acc_num, acc_type, balance):
        pass

    @abstractmethod
    def list_accounts(self):
        pass

    @abstractmethod
    def get_account_details(self, account_number):
        pass

class CustomerServiceProvider(ICustomerServiceProvider):
    db = Queryconnection(host="localhost", user="root", password="root",
port="3306", database="ASSIGN1")

    # Create a new account
    db.create_account("Jeremy", 12, "savings", 1000.0)
    db.create_account("Kumar", 48, "current", 14000.0)

    # List all accounts
    accounts = db.list_accounts()
    print("All accounts:", accounts)
    print()

    # Get account details
    printing = db.get_account_details(12)

    print()
    print("Account details:", printing)
    db.close_connection()
import mysql.connector

class Queryconnection:
    def __init__(self, host, user, password, port, database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()

    def create_account(self, customer, acc_num, acc_type, balance):
        query = "INSERT INTO customerserviceprovider (customer, acc_num,
acc_type, balance) VALUES (%s, %s, %s,%s)"
        values = (customer, acc_num, acc_type, balance)
        self.cursor.execute(query, values)
        self.connection.commit()

    def list_accounts(self):
        self.cursor.execute("SELECT * FROM customerserviceprovider")
        accounts = self.cursor.fetchall()
        return accounts
```

```
    def get_account_details(self, account_number):
        query = "SELECT * FROM customerserviceprovider WHERE acc_num = %s"
        self.cursor.execute(query, (account_number,))
        account_details = self.cursor.fetchone()
        return account_details

    def close_connection(self):
        self.connection.close()
```

	customer_id	customer_name	account_type	balance
▶	1	Jeremy	Savings	5000
	2	Kumar	Current	10000
	NULL	NULL	NULL	NULL

```
All Accounts: [(1,'Jeremy','savings',5000)], [(2,'Kumar','current',10000)]

Account details:(1,'Jeremy','savings',5000)
```