BANKING SYSTEM

A Python Project Submitted

In

COMPUTER SCIENCE AND ENGINEERING

by

SUDHANSHU SINGH (Roll No. 2401330100369)

HARSH RAJ(Roll No. 2401330100170)

SHRIDHAR PANDEY (Roll No. 2401330100349)

Under the Supervision of

Mr. MOHIT KUMAR
Assistance Professor ,CSE , NIET

Ms. MANSHI CHAUDHARY
Assistance Professor ,MCA, NIET

Department of Computer Science And Engineering
NOIDA INSTITUTE OF ENGINEERING AND TECHNOLOGY,
GREATER NOIDA
(An Autonomous Institute)
Affiliated to
DR. A.P.J. ABDUL KALAM TECHNICAL UNIVERSITY, LUCKNOW
June, 2025

Banking System

1. Project Description

Develop a console-based application to simulate a simplified Banking System. This system will allow an administrator (user of the application) to:

- Manage customer records (add, remove, view).
- Create different types of bank accounts (Savings, Checking) for customers.
- Perform basic banking operations: deposit, withdraw, and transfer funds between accounts.
- View account details and customer's associated accounts.
- Apply interest to savings accounts.
- Handle overdraft limits for checking accounts.
- Persist all banking data (customers, accounts, balances, relationships) to files so it's saved between application runs.

This project heavily utilizes **Object-Oriented Programming (OOP)** principles, particularly **inheritance** to model different account types, **polymorphism** for transaction operations, and **encapsulation** for data integrity.

2. Core OOP Concepts to Apply

- **Class & Object Design:** Model Customer, Account(and its specialized types), and the Bank that orchestrates everything.
- **Encapsulation:** Use private-like attributes (e.g., _balance, _overdraft_limit) and expose them via public methods or @propertydecorators for controlled access and data integrity.
- **Constructors (__init__):** Properly initialize object states.
- **Instance Methods:** Define behaviors specific to an account, customer, or the bank.
- **Abstract Base Classes (abcmodule):** Use ABCand @abstractmethodfor the base Accountclass to enforce that all account types implement deposit()and withdraw().
- **Inheritance:** Create SavingsAccountand CheckingAccountclasses inheriting from the base Account class.
- **Polymorphism:** Call the deposit()or withdraw()methods on various Account objects (different types) through a common interface, demonstrating different behaviors (e.g., overdraft check for checking).
- **Object Composition:** The Bankclass will *contain* collections of Customerand Accountobjects. Customerobjects will *contain* a list of associated account *numbers* (for persistence).

- **Magic Methods (__str__, __repr__):** Implement these for clear object representation.
- **Error Handling:** Use try-exceptblocks for robust user input validation and transaction failures (e.g., insufficient funds).

3. Class Specifications & Coder Tasks

You are required to implement the following classes with the specified attributes and methods.

3.1. AccountClass (Abstract Base Class)

Represents a generic bank account. All specific account types will inherit from this class.

- **Attributes (Private-like for encapsulation):**
  - _account_number: str- A unique identifier for the account.
  - _balance: float- The current balance in the account (default 0.0).
  - _account_holder_id: str- The ID of the Customerwho owns this account.
- **Properties:**
  - account_number(read-only): Returns _account_number.
  - balance(read-only): Returns _balance.
  - account_holder_id(read-only): Returns _account_holder_id.
- **Methods:**
  - ___init_____(self, account_number: str, account_holder_id: str, initial_balance: float = 0.0):
    - Constructor. Initializes _account_number, _account_holder_id, and _balance.
  - deposit(self, amount: float) -> bool: **(Abstract Method)**
    - This method *must* be implemented by all concrete subclasses. It should add amount to _balance. Returns True on success, False on failure (e.g., negative amount).
  - withdraw(self, amount: float) -> bool: **(Abstract Method)**
    - This method *must* be implemented by all concrete subclasses. It should attempt to subtract amount from _balance. Returns True on success, Falseon failure (e.g., insufficient funds, negative amount).
  - display_details(self) -> str:
    - Returns a basic string representation: "Acc No: [Number], Balance: $[Balance]". This method will be extended/overridden in child classes.
  - to_dict(self) -> dict:

- Returns a dictionary representation of the account's basic attributes, useful for saving to file. Include a typekey (e.g., 'savings', 'checking') to aid deserialization.

3.2. SavingsAccountClass

Inherits from Account, representing a savings account with interest.

- **Attributes (Specific to SavingsAccount):**
  - _interest_rate: float- Annual interest rate (e.g., 0.01 for 1%).
- **Properties:**
  - interest_rate(read/write): Returns/sets _interest_rate. Setter enforces non-negative.
- **Methods:**
  - __init__(self, account_number: str, account_holder_id: str, initial_balance: float = 0.0, interest_rate: float = 0.01):
    - Constructor. Calls parent __init__and initializes _interest_rate.
  - deposit(self, amount: float) -> bool:
    - **Implements** abstract method. Adds amount if valid. Returns Trueon success, Falseotherwise.
  - withdraw(self, amount: float) -> bool:
    - **Implements** abstract method. Withdraws amount if sufficient balance (no overdraft allowed). Returns Trueon success, Falseotherwise.
  - apply_interest(self) -> None:
    - Calculates and adds interest to the balance (_balance += _balance * _interest_rate).
  - display_details(self) -> str:
    - Extends/Overrides parent. Returns a string including inherited details plus interest rate.
  - to_dict(self) -> dict:
    - Extends parent. Returns dictionary including _interest_rateand type: 'savings'.

3.3. CheckingAccountClass

Inherits from Account, representing a checking account with an overdraft limit.

- **Attributes (Specific to CheckingAccount):**
  - _overdraft_limit: float- The maximum negative balance allowed.
- **Properties:**
  - overdraft_limit(read/write): Returns/sets _overdraft_limit. Setter enforces non-negative.

- **Methods:**
  - __init__(self, account_number: str, account_holder_id: str, initial_balance: float = 0.0, overdraft_limit: float = 0.0):
    - Constructor. Calls parent __init__and initializes _overdraft_limit.
  - deposit(self, amount: float) -> bool:
    - **Implements** abstract method. Adds amount if valid. Returns Trueon success, Falseotherwise.
  - withdraw(self, amount: float) -> bool:
    - **Implements** abstract method. Withdraws amount if _balance - amount >= -_overdraft_limit. Returns Trueon success, False otherwise.
  - display_details(self) -> str:
    - Extends/Overrides parent. Returns a string including inherited details plus overdraft limit.
  - to_dict(self) -> dict:
    - Extends parent. Returns dictionary including _overdraft_limitand type: 'checking'.

3.4. CustomerClass

Represents a bank customer.

- **Attributes (Private-like for encapsulation):**
  - _customer_id: str- A unique identifier for the customer.
  - _name: str- The customer's full name.
  - _address: str- The customer's address.
  - _account_numbers: list[str]- A list of account numbers (strings) associated with this customer.
- **Properties:**
  - customer_id(read-only): Returns _customer_id.
  - name(read-only): Returns _name.
  - address(read/write): Returns/sets _address.
  - account_numbers(read-only): Returns a copy of _account_numbers.
- **Methods:**
  - __init__(self, customer_id: str, name: str, address: str):
    - Constructor. Initializes attributes. _account_numbersstarts empty.
  - add_account_number(self, account_number: str) -> None:
    - Adds an account number to _account_numbersif not already present.
  - remove_account_number(self, account_number: str) -> None:
    - Removes an account number from _account_numbers.

- display_details(self) -> str:
  - Returns a string including customer ID, name, address, and number of accounts.
- to_dict(self) -> dict:
  - Returns a dictionary representation of the customer's attributes, useful for saving to file.

3.5. BankClass

The main orchestrator class, managing all Customerand Accountobjects.

- **Attributes (Private-like for encapsulation):**
  - _customers: dict[str, Customer]- Dictionary: customer_id-> Customerobject.
  - _accounts: dict[str, Account]- Dictionary: account_number-> Accountobject.
  - _customer_file: str- Filename for customer data (e.g., 'customers.json').
  - _account_file: str- Filename for account data (e.g., 'accounts.json').
- **Methods:**
  - __init__(self,customer_file='customers.json', account_file='accounts.json'):
    - Initializes empty dictionaries. Sets file names.
    - Calls _load_data()to load existing data from files.
  - _load_data(self) -> None: (Private Helper Method)
    - Loads customer data from _customer_file.
    - Loads account data from _account_file.
    - Handles FileNotFoundError.
    - **Crucial for Relationships:** After loading raw dictionary data, you must iterate through them to create actual Customerand Account*objects*. For accounts, use the typefield to instantiate SavingsAccountor CheckingAccount. Ensure that accounts loaded are correctly referenced by their account_holder_idin the _customersdictionary.
  - _save_data(self) -> None: (Private Helper Method)
    - Saves current _customersand _accountsdata to their respective JSON files.
    - Iterate through collections and call to_dict()on each object before saving.
  - add_customer(self, customer: Customer) -> bool:
    - Adds a Customerobject. Returns Trueif added, Falseif ID exists. Calls _save_data().

- remove_customer(self, customer_id: str) -> bool:
  - Removes a customer. Returns Trueif removed, Falseif not found or if customer has active accounts. Calls _save_data().
- create_account(self, customer_id: str, account_type: str, initial_balance: float = 0.0, **kwargs) -> Account | None:
  - Creates a new account (Savings or Checking) for an existing customer.
  - Generates a unique account_number(e.g., simple incremental string or UUID).
  - Instantiates the correct Accountsubclass.
  - Adds the new account to _accountsand associates it with the customer by adding its number to customer.add_account_number().
  - Returns the new Accountobject on success, Noneon failure (e.g., customer not found, invalid account type). Calls _save_data().
- deposit(self, account_number: str, amount: float) -> bool:
  - Finds the account by account_number. Calls account.deposit(amount). Returns Trueon success, False otherwise. Calls _save_data().
- withdraw(self, account_number: str, amount: float) -> bool:
  - Finds the account by account_number. Calls account.withdraw(amount). Returns Trueon success, False otherwise. Calls _save_data().
- transfer_funds(self, from_acc_num: str, to_acc_num: str, amount: float) -> bool:
  - Withdraws from from_acc_numand deposits into to_acc_num. Ensures both accounts exist and transfer is valid.
  - Returns Trueon success, Falseotherwise. Calls _save_data().
- get_customer_accounts(self, customer_id: str) -> list[Account]:
  - Returns a list of Accountobjects associated with the given customer ID. Handles customer not found.
- display_all_customers(self) -> None:
  - Prints details of all registered customers.
- display_all_accounts(self) -> None:
  - Prints details of all accounts, using account.display_details().
- apply_all_interest(self) -> None: (Manager function)
  - Iterates through all SavingsAccountobjects and calls apply_interest(). Calls _save_data().

4. Coder Tasks (Implementation Steps)

1. **Set up Project Structure:** Create banking_system.py(or similar). Import abcand abstractmethod.
2. **Implement AccountClass (Abstract):** Define Accountas an ABCwith depositand withdraw as abstractmethods. Implement __init__, display_details, to_dict.
3. **Implement SavingsAccountand CheckingAccountClasses:** Implement these, inheriting from Account, and correctly implement their specific deposit(), withdraw(), display_details(), to_dict().
4. **Implement CustomerClass:** Write the Customerclass with all specified attributes, properties, and methods.
5. **Implement BankClass (Core Logic):**
    - Implement __init__, _load_data, _save_data. **Pay close attention to object reconstruction in _load_data:**
        - When loading accounts, use the typefield to instantiate SavingsAccountor CheckingAccount.
        - After loading both customers and accounts, ensure that Customer.account_numbers correctly reflects the account_numbers of existing accounts.
    - Implement add_customer, remove_customer. These should call _save_data().
    - Implement create_account. This is complex as it creates an Accountobject and links it to a Customer.
    - Implement deposit, withdraw, transfer_funds. These methods showcase polymorphism by calling deposit()/withdraw()on Accountobjects, which will behave differently based on the actual subclass.
    - Implement get_customer_accounts, display_all_customers, display_all_accounts, apply_all_interest.
6. **Create Console Interface:**
    - Develop a main()function or a run()method in the Bankclass.
    - Implement a main whileloop that presents a menu (e.g., "1. Add Customer", "2. Create Account", "3. Deposit", "4. Withdraw", "5. Transfer", "6. View Customer Accounts", "7. Apply Interest", "8. Exit").
    - Use input()to get choices and data.
    - Call the appropriate Bankmethods.
    - Include clear print()statements for feedback and displaying results.

CODE:

```
# banking_system.py
# Abstract Base Class

class Account:
    def __init__(self, account_number, account_holder_id,
initial_balance=0.0):
        self._account_number = account_number
        self._account_holder_id = account_holder_id
        self._balance = initial_balance

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            print(f"Deposit successful. Current Balance:
₹{self._balance}")
            return True
        else:
            print("Invalid deposit amount.")
            return False

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
            print(f"Withdrawal successful. Current Balance:
₹{self._balance}")
            return True
        else:
            print("Insufficient funds or invalid amount.")
```

```
            return False

    def display_details(self):
        return f"Account No: {self._account_number}, Balance:
₹{self._balance}"

    def get_account_number(self):
        return self._account_number

    def get_holder_id(self):
        return self._account_holder_id

    def get_balance(self):
        return self._balance

class SavingsAccount(Account):
    def __init__(self, account_number, account_holder_id,
initial_balance=0.0, interest_rate=0.01):
        super().__init__(account_number, account_holder_id,
initial_balance)
        self._interest_rate = interest_rate

    def apply_interest(self):
        interest = self._balance * self._interest_rate
        self._balance += interest
        print(f"Interest applied. New Balance: ₹{self._balance}")

    def display_details(self):
        return super().display_details() + f", Interest Rate:
{self._interest_rate * 100}%"

class CheckingAccount(Account):
```

```
    def __init__(self, account_number, account_holder_id,
initial_balance=0.0, overdraft_limit=0.0):
        super().__init__(account_number, account_holder_id,
initial_balance)
        self._overdraft_limit = overdraft_limit

    def withdraw(self, amount):
        if 0 < amount <= self._balance + self._overdraft_limit:
            self._balance -= amount
            print(f"Withdrawal successful. Current Balance:
₹{self._balance}")
            return True
        else:
            print("Overdraft limit exceeded or invalid amount.")
            return False

    def display_details(self):
        return super().display_details() + f", Overdraft Limit:
₹{self._overdraft_limit}"

class Customer:
    def __init__(self, customer_id, name, address):
        self._customer_id = customer_id
        self._name = name
        self._address = address
        self._accounts = []

    def add_account(self, account_number):
        if account_number not in self._accounts:
            self._accounts.append(account_number)

    def get_id(self):
```

```
        return self._customer_id

    def display_details(self):
        return f"Customer ID: {self._customer_id}, Name:
{self._name}, Address: {self._address}, Accounts:
{self._accounts}"

    def get_accounts(self):
        return self._accounts

class Bank:
    def __init__(self):
        self._customers = {}
        self._accounts = {}
        self._admin_password = "admin123"

    def create_customer_and_account(self):
        customer_id = input("Enter Customer ID: ")
        if customer_id in self._customers:
            print("Customer already exists. Cannot proceed.")
            return

        name = input("Enter Name: ")
        address = input("Enter Address: ")
        acc_number = input("Enter unique Account Number: ")

        if acc_number in self._accounts:
            print("Account already exists with this number.")
            return

        acc_type = input("Enter account type (savings/checking): ")
```

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

        if acc_type == "savings":
            acc = SavingsAccount(acc_number, customer_id, balance)
        elif acc_type == "checking":
            overdraft = float(input("Enter overdraft limit: "))
            acc = CheckingAccount(acc_number, customer_id, balance,
overdraft)
        else:
            print("Invalid account type.")
            return

        self._customers[customer_id] = Customer(customer_id, name,
address)
        self._accounts[acc_number] = acc
        self._customers[customer_id].add_account(acc_number)
        print("Customer and Account created successfully.")

    def show_all_customers(self):
        pw = input("Enter admin password: ")
        if pw == self._admin_password:
            for cust in self._customers.values():
                print(cust.display_details())
        else:
            print("Access Denied: Not Admin.")

    def show_all_accounts(self):
        pw = input("Enter admin password: ")
        if pw == self._admin_password:
            for acc in self._accounts.values():
                print(acc.display_details())
```

```
        else:
            print("Access Denied: Not Admin.")

    def deposit(self):
        acc_num = input("Enter Account Number: ")
        amount = float(input("Enter amount to deposit: "))
        if acc_num in self._accounts:
            self._accounts[acc_num].deposit(amount)
        else:
            print("Account not found.")

    def withdraw(self):
        acc_num = input("Enter Account Number: ")
        amount = float(input("Enter amount to withdraw: "))
        if acc_num in self._accounts:
            self._accounts[acc_num].withdraw(amount)
        else:
            print("Account not found.")

    def transfer(self):
        from_acc = input("Enter FROM Account: ")
        to_acc = input("Enter TO Account: ")
        amount = float(input("Enter amount to transfer: "))
        if from_acc in self._accounts and to_acc in self._accounts:
            if self._accounts[from_acc].withdraw(amount):
                self._accounts[to_acc].deposit(amount)
        else:
            print("One or both account numbers not found.")

    def apply_interest_to_all(self):
        for acc in self._accounts.values():
```

```
            if isinstance(acc, SavingsAccount):
                acc.apply_interest()

def main():
    bank = Bank()
    while True:
        print("\n--- BANK MENU ---")
        print("1. Create Customer and Account")
        print("2. Deposit")
        print("3. Withdraw")
        print("4. Transfer Funds")
        print("5. Show All Customers")
        print("6. Show All Accounts")
        print("7. Apply Interest to Savings Accounts")
        print("8. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            bank.create_customer_and_account()
        elif choice == '2':
            bank.deposit()
        elif choice == '3':
            bank.withdraw()
        elif choice == '4':
            bank.transfer()
        elif choice == '5':
            bank.show_all_customers()
        elif choice == '6':
            bank.show_all_accounts()
        elif choice == '7':
            bank.apply_interest_to_all()
```

```
        elif choice == '8':
            break
        else:
            print("Invalid choice.")

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

OUTPUTS:

1.Main : ---BANK MENU---

```
--- BANK MENU ---
1. Create Customer and Account
2. Deposit
3. Withdraw
4. Transfer Funds
5. Show All Customers
6. Show All Accounts
7. Apply Interest to Savings Accounts
8. Exit
Enter your choice:  1
Enter Customer ID:  ↑↓ for history. Search history with c-↑/c-↓
```

2. Create Customer and Accounts :

```
Enter your choice:  1
Enter Customer ID:  1
Enter Name:  Harsh
Enter Address:  Alpha 2
Enter unique Account Number:  100
Enter account type (savings/current):  savings
Enter initial balance:  5000
Customer and Account created successfully.
```

```
Enter your choice:  1
Enter Customer ID:  2
Enter Name:  Sudhanshu
Enter Address:  Varanasi
Enter unique Account Number:  200
Enter account type (savings/current):  savings
Enter initial balance:  2000
Customer and Account created successfully.
```

3. Deposit :

```
Enter your choice:  2
Enter Account Number:  100
Enter amount to deposit:  5000
Deposit successful. Current Balance: ₹10000.0
```

4. Withdraw :

```
Enter your choice:  3
Enter Account Number:  100
Enter amount to withdraw:  2000
Withdrawal successful. Current Balance: ₹8000.0
```

5. Transfer Funds :

```
Enter your choice:  4
Enter FROM Account:  100
Enter TO Account:  200
Enter amount to transfer:  3000
Withdrawal successful. Current Balance: ₹5000.0
Deposit successful. Current Balance: ₹5000.0
```

6. Show All Customers :

```
Enter your choice:  5
Enter admin password:  admin123
Customer ID: 1, Name: Harsh, Address: Alpha 2, Accounts: ['100']
Customer ID: 2, Name: Sudhanshu, Address: Varanasi, Accounts: ['200']
```

7. Show All Accounts :

```
Enter your choice:  6
Enter admin password:  admin123
Account No: 100, Balance: ₹5000.0, Interest Rate: 1.0%
Account No: 200, Balance: ₹5000.0, Interest Rate: 1.0%
```

8. Apply Interest to Savings Accounts :

```
Enter your choice:  7
Interest applied. New Balance: ₹5050.0
Interest applied. New Balance: ₹5050.0
```

9. Exit :

```
Enter your choice:  8
```