Assignment 5

Create an Account class Heirarchy

Account with super class (acc_id, name, balance)

methods - withdraw and deposit

In [10]:

```
class Account:
    """Superclass for bank accounts with basic transactions."""
    def __init__(self, acc_id, name, balance=0.0):
        self.acc_id = acc_id
        self.name = name
        self.balance = max(0.0, float(balance))

    def __str__(self):
        return f"ID: {self.acc_id}, Name: {self.name}, Balance: Rs.{self.balance:,.

    def deposit(self, amount):
        """Adds amount to the balance."""
        amount = float(amount)
        if amount > 0:
            self.balance += amount
            print(f"💰 Deposited Rs.{amount:,.2f}. New Balance: Rs.{self.balance:,.
            return True
        print("❌ Deposit amount must be positive.")
        return False

    def withdraw(self, amount):
        """Removes amount from the balance if sufficient funds exist."""
        amount = float(amount)
        if amount <= 0:
            print("❌ Withdrawal amount must be positive.")
            return False

        if self.balance >= amount:
            self.balance -= amount
            print(f"💸 Withdrew Rs.{amount:,.2f}. New Balance: Rs.{self.balance:,.2
            return True
        else:
            print(f"🛑 Insufficient funds. Available: Rs.{self.balance:,.2f}.")
            return False

# --- Subclass Example (Optional but shows Hierarchy) ---

class SavingsAccount(Account):
    """Subclass adding an interest rate attribute."""
    def __init__(self, acc_id, name, balance=0.0, interest_rate=0.01):
        super().__init__(acc_id, name, balance)
        self.interest_rate = interest_rate

    def apply_interest(self):
        """Adds interest to the balance."""
        interest = self.balance * self.interest_rate
```

```
        self.balance += interest
        print(f"⭐ Interest applied. Amount: Rs.{interest:,.2f}")
```

In [11]:
```
# Create accounts
acc = Account(1001, "Jane Doe", 500)
sav = SavingsAccount(2002, "John Smith", 1000, 0.03)

print("\nInitial State:")
print(acc)
print(sav)

# Test transactions
print("\n--- Testing Transactions ---")
acc.deposit(250)    # Success
acc.withdraw(100)   # Success
acc.withdraw(800)   # Failure (Insufficient funds)

# Test Subclass method
print("\n--- Testing Savings ---")
sav.deposit(500)
sav.apply_interest()

print("\nFinal State:")
print(acc)
print(sav)
```

```
Initial State:
ID: 1001, Name: Jane Doe, Balance: Rs.500.00
ID: 2002, Name: John Smith, Balance: Rs.1,000.00

--- Testing Transactions ---
💰 Deposited Rs.250.00. New Balance: Rs.750.00
💸 Withdrew Rs.100.00. New Balance: Rs.650.00
🛑 Insufficient funds. Available: Rs.650.00.

--- Testing Savings ---
💰 Deposited Rs.500.00. New Balance: Rs.1,500.00
⭐ Interest applied. Amount: Rs.45.00

Final State:
ID: 1001, Name: Jane Doe, Balance: Rs.650.00
ID: 2002, Name: John Smith, Balance: Rs.1,545.00
```

Create SavingsAccount as sub class of account - additional field (type - personal/corporate etc) implement withdraw and deposit such that

- maximum upto 1 lakh can be deposited in an account at a time
- Min balance 5000 must be maintained while withdrawal (if type = corporate you withdraw full amount = balance)

In [12]:
```
class SavingsAccount(Account):
    """
    Subclass of Account with specific rules for deposit limit and minimum balance.
    Additional field: type (personal/corporate, etc.)
    """
```

```
    MAX_DEPOSIT_LIMIT = 100000.00
    MIN_PERSONAL_BALANCE = 5000.00

    def __init__(self, acc_id, name, type, balance=0.0):
        # type should be case-insensitive for easier logic
        valid_types = ['personal', 'corporate']
        self.type = type.lower() if type.lower() in valid_types else 'personal'
        super().__init__(acc_id, name, balance)
        print(f"   (Account Type: {self.type.capitalize()} Savings)")

    def __str__(self):
        """Overrides string method to include account type."""
        base_str = super().__str__()
        return f"{base_str}, Type: {self.type.capitalize()}"

    # --- Custom Deposit Implementation ---
    def deposit(self, amount):
        """
        Overrides deposit. Max up to 1 lakh can be deposited at a time.
        """
        try:
            amount = float(amount)
            if amount <= 0:
                print("❌ Deposit failed. Amount must be positive.")
                return False

            # Rule 1: Maximum 1 lakh deposit limit
            if amount > self.MAX_DEPOSIT_LIMIT:
                print(f"🛑 Deposit limit exceeded. Max amount allowed is Rs.{self.M
                return False

            # If rules pass, call the superclass's deposit method
            return super().deposit(amount)

        except ValueError:
            print("❌ Invalid input. Deposit amount must be a number.")
            return False

    # --- Custom Withdraw Implementation ---
    def withdraw(self, amount):
        """
        Overrides withdraw. Enforces min balance of Rs.5000 (unless Corporate).
        """
        try:
            amount = float(amount)
            if amount <= 0:
                print("❌ Withdrawal failed. Amount must be positive.")
                return False

            remaining_balance = self.balance - amount

            if self.type == 'corporate':
                # Rule 2 (Corporate): You can withdraw the full amount (down to 0)
                if remaining_balance >= 0:
                    return super().withdraw(amount)
                else:
```

```
                    print("🛑 Withdrawal failed. Corporate account must have suffic
                    return False

            elif self.type == 'personal':
                # Rule 2 (Personal): Min balance 5000 must be maintained
                if remaining_balance >= self.MIN_PERSONAL_BALANCE:
                    return super().withdraw(amount)
                else:
                    print(f"🛑 Withdrawal failed. Personal accounts must maintain a
                    print(f"   Max you can withdraw is Rs.{self.balance - self.MIN_
                    return False

            return False # Should not be reached

        except ValueError:
            print("❌ Invalid input. Withdrawal amount must be a number.")
            return False
```

In [13]:

```
# 1. Personal Account: Starts with Rs.10,000, must maintain Rs.5,000
p_acc = SavingsAccount(100, "Alice (Personal)", "personal", 10000.00)
# 2. Corporate Account: Starts with Rs.10,000, no min balance
c_acc = SavingsAccount(200, "Global Corp", "corporate", 10000.00)

print("\n--- Initial Balances ---")
print(p_acc)
print(c_acc)

print("\n--- Test 1: Deposit Limit (1 Lakh / Rs.100,000) ---")
p_acc.deposit(99000.00)  # ✅ Success (Below limit)
p_acc.deposit(100000.01) # 🛑 Failure (Above limit)

print("\n--- Test 2: Personal Account Withdrawal (Min Rs.5,000 Rule) ---")
# Current balance: Rs.10,000 + Rs.99,000 = Rs.109,000
p_acc.withdraw(104000.00) # ✅ Success (Remaining balance: Rs.5,000)
p_acc.withdraw(1.00)       # 🛑 Failure (Remaining balance would be Rs.4,999, below

print("\n--- Test 3: Corporate Account Withdrawal (Full Withdrawal Allowed) ---")
# Corporate balance is Rs.10,000
c_acc.withdraw(9999.99) # ✅ Success (Remaining balance: Rs.0.01, allowed for corpo
c_acc.withdraw(0.01)     # ✅ Success (Remaining balance: Rs.0.00)

print("\n--- Final Balances ---")
print(p_acc)
print(c_acc)
```

```
   (Account Type: Personal Savings)
   (Account Type: Corporate Savings)

--- Initial Balances ---
ID: 100, Name: Alice (Personal), Balance: Rs.10,000.00, Type: Personal
ID: 200, Name: Global Corp, Balance: Rs.10,000.00, Type: Corporate

--- Test 1: Deposit Limit (1 Lakh / Rs.100,000) ---
💰 Deposited Rs.99,000.00. New Balance: Rs.109,000.00
🛑 Deposit limit exceeded. Max amount allowed is Rs.100,000.00.

--- Test 2: Personal Account Withdrawal (Min Rs.5,000 Rule) ---
💸 Withdrew Rs.104,000.00. New Balance: Rs.5,000.00
🛑 Withdrawal failed. Personal accounts must maintain a min balance of Rs.5,000.00.
   Max you can withdraw is Rs.0.00.

--- Test 3: Corporate Account Withdrawal (Full Withdrawal Allowed) ---
💸 Withdrew Rs.9,999.99. New Balance: Rs.0.01
💸 Withdrew Rs.0.01. New Balance: Rs.0.00

--- Final Balances ---
ID: 100, Name: Alice (Personal), Balance: Rs.5,000.00, Type: Personal
ID: 200, Name: Global Corp, Balance: Rs.0.00, Type: Corporate
```

Create CurrentAccount as sub class of account implement withdraw and deposit such that

- maximum upto 2 lakh can be deposited in an account at a time
- Min balance 10000 must be maintained while withdrawal

In [14]:

```python
class CurrentAccount(Account):
    """
    Subclass of Account with specific rules for deposit limit and minimum balance.
    """
    # Define class constants for the rules (assuming values in Indian Rupees)
    MAX_DEPOSIT_LIMIT = 200000.00  # ₹2 Lakh
    MIN_BALANCE = 10000.00         # ₹10 Thousand

    def __init__(self, acc_id, name, balance=0.0):
        super().__init__(acc_id, name, balance)
        print(f"✅ Current Account {self.acc_id} created for {self.name}.")

    def __str__(self):
        """Includes account type in the description."""
        base_str = super().__str__()
        return f"{base_str} | Type: Current"

    # --- Custom Deposit Implementation ---
    def deposit(self, amount):
        """
        Overrides deposit. Maximum ₹2,00,000 can be deposited at a time.
        """
        try:
            amount = float(amount)
            if amount <= 0:
                print("❌ Deposit failed. Amount must be positive.")
```

```
                return False

            # Rule 1: Maximum deposit limit enforcement
            if amount > self.MAX_DEPOSIT_LIMIT:
                print(f"🛑 Deposit limit exceeded! Max allowed is ₹{self.MAX_DEPOSI
                return False

            # If rules pass, call the superclass method and provide user feedback
            if super().deposit(amount):
                print(f"💰 Deposited ₹{amount:,.2f}. New Balance: ₹{self.balance:,.
                return True
            return False

        except ValueError:
            print("❌ Invalid input. Deposit amount must be a number.")
            return False

    # --- Custom Withdraw Implementation ---
    def withdraw(self, amount):
        """
        Overrides withdraw. Enforces a minimum balance of ₹10,000.
        """
        try:
            amount = float(amount)
            if amount <= 0:
                print("❌ Withdrawal failed. Amount must be positive.")
                return False

            remaining_balance = self.balance - amount

            # Rule 2: Minimum balance maintenance check
            if remaining_balance < self.MIN_BALANCE:
                print(f"🛑 Withdrawal failed. You must maintain a min balance of ₹
                max_withdraw = max(0, self.balance - self.MIN_BALANCE)
                print(f"   Max you can withdraw is ₹{max_withdraw:,.2f}.")
                return False

            # If rules pass, call the superclass method and provide user feedback
            if super().withdraw(amount):
                print(f"💸 Withdrew ₹{amount:,.2f}. New Balance: ₹{self.balance:,.2
                return True

            # Should not hit this return unless super() fails for other reasons (e.
            print("🛑 Withdrawal failed due to insufficient funds.")
            return False

        except ValueError:
            print("❌ Invalid input. Withdrawal amount must be a number.")
            return False
```

In [15]:

```
# Create a Current Account starting with ₹50,000
current_acc = CurrentAccount(301, "Acme Solutions", 50000.00)

print("\n--- Initial Balance ---")
print(current_acc)
```

```
print("\n--- Test 1: Deposit Limit (Max ₹2,00,000) ---")
# Success: Below the limit
current_acc.deposit(150000.00)
# Failure: Above the limit
current_acc.deposit(200000.01)

print("\n--- Test 2: Withdrawal (Min Balance ₹10,000 Rule) ---")
# Current Balance is ₹50,000 + ₹1,50,000 = ₹2,00,000

# Success: Leaves exactly ₹10,000 balance
current_acc.withdraw(190000.00)

# Failure: Would leave a balance of ₹9,999 (below ₹10,000 min)
current_acc.withdraw(1.00)

# Success: Max allowed withdrawal
current_acc.withdraw(0.00) # (No change, but technically allowed)

print("\n--- Final Balance ---")
print(current_acc)
```

```
✅ Current Account 301 created for Acme Solutions.

--- Initial Balance ---
ID: 301, Name: Acme Solutions, Balance: Rs.50,000.00 | Type: Current

--- Test 1: Deposit Limit (Max ₹2,00,000) ---
💰 Deposited Rs.150,000.00. New Balance: Rs.200,000.00
💰 Deposited ₹150,000.00. New Balance: ₹200,000.00
🛑 Deposit limit exceeded! Max allowed is ₹200,000.00 in a single transaction.

--- Test 2: Withdrawal (Min Balance ₹10,000 Rule) ---
💸 Withdrew Rs.190,000.00. New Balance: Rs.10,000.00
💸 Withdrew ₹190,000.00. New Balance: ₹10,000.00
🛑 Withdrawal failed. You must maintain a min balance of ₹10,000.00.
   Max you can withdraw is ₹0.00.
❌ Withdrawal failed. Amount must be positive.

--- Final Balance ---
ID: 301, Name: Acme Solutions, Balance: Rs.10,000.00 | Type: Current
```

Create Bank App with Transaction class Create Method withdraw_from_account(account : Account) and deposit_to_account(account : Account) These methods will return the new balance after deposite/withdraw

In [17]:

```
import datetime

# --- 1. Base Account Hierarchy (Refined for Rs. currency and method returns) ---

class Account:
    """Superclass representing a generic bank account."""
    def __init__(self, acc_id, name, balance=0.0):
        self.acc_id = acc_id
        self.name = name
        self.balance = max(0.0, float(balance))
```

```
    def __str__(self):
        return f"ID: {self.acc_id}, Name: {self.name}, Balance: Rs.{self.balance:,.

    def deposit(self, amount):
        """Basic deposit: Adds funds and returns True on success."""
        amount = float(amount)
        if amount > 0:
            self.balance += amount
            return True
        return False

    def withdraw(self, amount):
        """Basic withdraw: Removes funds if available and returns True on success."
        amount = float(amount)
        if amount > 0 and self.balance >= amount:
            self.balance -= amount
            return True
        return False

class SavingsAccount(Account):
    """Subclass with specific rules for minimum balance and deposit limit."""
    MAX_DEPOSIT_LIMIT = 100000.00
    MIN_PERSONAL_BALANCE = 5000.00

    def __init__(self, acc_id, name, type, balance=0.0):
        valid_types = ['personal', 'corporate']
        self.type = type.lower() if type.lower() in valid_types else 'personal'
        super().__init__(acc_id, name, balance)

    def __str__(self):
        base_str = super().__str__()
        return f"{base_str}, Type: Savings ({self.type.capitalize()})"

    def deposit(self, amount):
        """Max Rs. 1 Lakh can be deposited at a time."""
        amount = float(amount)
        if amount > self.MAX_DEPOSIT_LIMIT:
            print(f"🛑 Error: Deposit limit exceeded. Max allowed is Rs.{self.MAX_I
            return False
        return super().deposit(amount)

    def withdraw(self, amount):
        """Min balance Rs. 5000 must be maintained (unless corporate, then balance
        amount = float(amount)
        remaining_balance = self.balance - amount

        if self.type == 'corporate' and remaining_balance >= 0:
            return super().withdraw(amount)

        elif self.type == 'personal' and remaining_balance >= self.MIN_PERSONAL_BAL
            return super().withdraw(amount)

        else:
            if self.type == 'personal':
                max_withdraw = max(0, self.balance - self.MIN_PERSONAL_BALANCE)
```

```
                print(f"🛑 Error: Personal accounts must maintain Rs.{self.MIN_PERS
            else:
                print("🛑 Error: Corporate account insufficient funds.")
            return False

class CurrentAccount(Account):
    """Subclass with specific rules for minimum balance and deposit limit."""
    MAX_DEPOSIT_LIMIT = 200000.00  # Rs. 2 Lakh
    MIN_BALANCE = 10000.00         # Rs. 10 Thousand

    def __init__(self, acc_id, name, balance=0.0):
        super().__init__(acc_id, name, balance)

    def __str__(self):
        base_str = super().__str__()
        return f"{base_str} | Type: Current"

    def deposit(self, amount):
        """Max Rs. 2 Lakh can be deposited at a time."""
        amount = float(amount)
        if amount > self.MAX_DEPOSIT_LIMIT:
            print(f"🛑 Error: Deposit limit exceeded. Max allowed is Rs.{self.MAX_D
            return False
        return super().deposit(amount)

    def withdraw(self, amount):
        """Min balance Rs. 10,000 must be maintained."""
        amount = float(amount)
        remaining_balance = self.balance - amount

        if remaining_balance >= self.MIN_BALANCE:
            return super().withdraw(amount)
        else:
            max_withdraw = max(0, self.balance - self.MIN_BALANCE)
            print(f"🛑 Error: Min balance of Rs.{self.MIN_BALANCE:,.2f} required. M
            return False

# --- 2. Transaction Class ---

class Transaction:
    """Stores details for a single account transaction."""
    def __init__(self, acc_id, type, amount, new_balance):
        self.acc_id = acc_id
        self.type = type
        self.amount = amount
        self.new_balance = new_balance
        self.timestamp = datetime.datetime.now()

    def __str__(self):
        return (f"[{self.timestamp.strftime('%Y-%m-%d %H:%M:%S')}] "
                f"Acc ID: {self.acc_id} | Type: {self.type} | "
                f"Amount: Rs.{self.amount:,.2f} | New Balance: Rs.{self.new_balance

# --- 3. BankApp Class ---

class BankApp:
```

```
    """Manages a collection of accounts and handles transactions."""
    def __init__(self):
        self.accounts = {}
        self.transactions = []
        self._next_id = 1001

    def add_account(self, account):
        """Adds a new account to the manager."""
        self.accounts[account.acc_id] = account
        self._next_id += 1

    def get_next_id(self):
        return self._next_id

    def find_account(self, acc_id):
        """Utility to retrieve an account by ID."""
        return self.accounts.get(acc_id)

    # --- Requested Method 1 ---
    def withdraw_from_account(self, account: Account) -> float:
        """
        Handles user input for withdrawal, executes transaction, logs it, and retur
        """
        print(f"\n--- Withdrawal for Account {account.acc_id} ({account.name}) ---"
        print(f"Current Balance: Rs.{account.balance:,.2f}")
        try:
            amount = float(input("Enter withdrawal amount (Rs.): "))
        except ValueError:
            print("❌ Invalid input. Please enter a numerical amount.")
            return account.balance

        if account.withdraw(amount):
            # Log successful transaction
            self.transactions.append(Transaction(account.acc_id, "Withdrawal", amou
            print(f"✅ Withdrawal successful. New Balance: Rs.{account.balance:,.2
            return account.balance
        else:
            # Failure message handled by account.withdraw() or above
            return account.balance

    # --- Requested Method 2 ---
    def deposit_to_account(self, account: Account) -> float:
        """
        Handles user input for deposit, executes transaction, logs it, and returns
        """
        print(f"\n--- Deposit for Account {account.acc_id} ({account.name}) ---")
        print(f"Current Balance: Rs.{account.balance:,.2f}")
        try:
            amount = float(input("Enter deposit amount (Rs.): "))
        except ValueError:
            print("❌ Invalid input. Please enter a numerical amount.")
            return account.balance

        if account.deposit(amount):
            # Log successful transaction
            self.transactions.append(Transaction(account.acc_id, "Deposit", amount,
```

```
            print(f"✅ Deposit successful. New Balance: Rs.{account.balance:,.2f}")
            return account.balance
        else:
            # Failure message handled by account.deposit() or above
            return account.balance

# --- DEMONSTRATION MENU ---

def main_menu():
    """Simple menu to test the BankApp functionalities."""
    bank = BankApp()

    # Initialize some accounts for testing
    bank.add_account(SavingsAccount(bank.get_next_id(), "Priya Sharma", "personal",
    bank.add_account(CurrentAccount(bank.get_next_id(), "Tech Solutions Pvt. Ltd.",
    bank.add_account(SavingsAccount(bank.get_next_id(), "Harsh Patel", "corporate",

    while True:
        print("\n======================================")
        print("  Gemini Bank Management System (Rs.)")
        print("======================================")
        print("1. Display All Accounts")
        print("2. Deposit Funds")
        print("3. Withdraw Funds")
        print("4. View Transaction History")
        print("5. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':
            print("\n--- All Active Accounts ---")
            if not bank.accounts:
                print("No accounts registered.")
                continue
            for acc in bank.accounts.values():
                print(acc)

        elif choice in ('2', '3'):
            try:
                acc_id = int(input("Enter Account ID: "))
                account = bank.find_account(acc_id)

                if account:
                    if choice == '2':
                        bank.deposit_to_account(account) # Returns new balance, but
                    else:
                        bank.withdraw_from_account(account) # Returns new balance
                else:
                    print("🛑 Account ID not found.")
            except ValueError:
                print("❌ Invalid input. Please enter a numerical ID.")

        elif choice == '4':
            print("\n--- Transaction History ---")
            if not bank.transactions:
                print("No transactions recorded yet.")
```

```
                continue
            for t in reversed(bank.transactions): # Show newest first
                print(t)

        elif choice == '5':
            print("Thank you for using the Gemini Bank System. Goodbye!")
            break

        else:
            print("❗ Invalid choice. Please select from 1 to 5.")

if __name__ == '__main__':
    main_menu()
```

```
====================================
  Gemini Bank Management System (Rs.)
====================================
1. Display All Accounts
2. Deposit Funds
3. Withdraw Funds
4. View Transaction History
5. Exit

--- All Active Accounts ---
ID: 1001, Name: Priya Sharma, Balance: Rs.50,000.00, Type: Savings (Personal)
ID: 1002, Name: Tech Solutions Pvt. Ltd., Balance: Rs.150,000.00 | Type: Current
ID: 1003, Name: Harsh Patel, Balance: Rs.75,000.00, Type: Savings (Corporate)

====================================
  Gemini Bank Management System (Rs.)
====================================
1. Display All Accounts
2. Deposit Funds
3. Withdraw Funds
4. View Transaction History
5. Exit

--- Deposit for Account 1001 (Priya Sharma) ---
Current Balance: Rs.50,000.00
✅ Deposit successful. New Balance: Rs.150,000.00

====================================
  Gemini Bank Management System (Rs.)
====================================
1. Display All Accounts
2. Deposit Funds
3. Withdraw Funds
4. View Transaction History
5. Exit

--- Transaction History ---
[2025-10-07 09:40:16] Acc ID: 1001 | Type: Deposit | Amount: Rs.100,000.00 | New Bal
ance: Rs.150,000.00

====================================
  Gemini Bank Management System (Rs.)
====================================
1. Display All Accounts
2. Deposit Funds
3. Withdraw Funds
4. View Transaction History
5. Exit
Thank you for using the Gemini Bank System. Goodbye!
```

Create user class with user interface that gives 2 menu options

1. Deposit
2. Withdraw

Both options will ask user to enter money to withdraw/deposite Display a statement with each transaction and final balance after user exits from the menu

In [18]:
```
class User:
    """Provides a menu-driven interface for a single account holder."""
    def __init__(self, account: Account, bank_app: BankApp):
        self.account = account
        self.bank_app = bank_app

    def _show_current_balance(self):
        """Prints the current account details."""
        print(f"\n[Current Balance for {self.account.name} (ID: {self.account.acc_i

    def run_menu(self):
        """
        Main loop for the user interface.
        Displays statement and final balance upon exiting.
        """
        print(f"\nWelcome, {self.account.name}! Accessing account ID: {self.account

        while True:
            self._show_current_balance()
            print("\n==========================")
            print("  User Transaction Menu")
            print("==========================")
            print("1. Deposit Money")
            print("2. Withdraw Money")
            print("3. Exit to Main Menu and View Statement")

            choice = input("Enter your choice (1-3): ")

            if choice == '1':
                self._handle_deposit()
            elif choice == '2':
                self._handle_withdraw()
            elif choice == '3':
                print("\n--- Exiting User Menu ---")
                self._display_final_statement()
                break
            else:
                print("❗ Invalid choice. Please select 1, 2, or 3.")

    def _handle_deposit(self):
        """Prompts for deposit amount and calls BankApp method."""
        try:
            amount = float(input("Enter amount to deposit (Rs.): "))
            if amount <= 0:
                print("❌ Deposit amount must be positive.")
                return

            initial_balance = self.account.balance
            new_balance = self.bank_app.deposit_to_account(self.account, amount)

            # Check if balance actually changed (deposit was successful based on su
            if new_balance > initial_balance:
```

```
                print(f"💰 Deposit successful. New Balance: Rs.{new_balance:,.2f}"
            # Failure message is handled by the Account method's print statement

        except ValueError:
            print("❌ Invalid input. Please enter a numerical amount.")

    def _handle_withdraw(self):
        """Prompts for withdrawal amount and calls BankApp method."""
        try:
            amount = float(input("Enter amount to withdraw (Rs.): "))
            if amount <= 0:
                print("❌ Withdrawal amount must be positive.")
                return

            initial_balance = self.account.balance
            new_balance = self.bank_app.withdraw_from_account(self.account, amount)

            # Check if balance actually changed (withdrawal was successful based on
            if new_balance < initial_balance:
                 print(f"💸 Withdrawal successful. New Balance: Rs.{new_balance:,.2
            # Failure message is handled by the Account method's print statement

        except ValueError:
            print("❌ Invalid input. Please enter a numerical amount.")

    def _display_final_statement(self):
        """Displays all transactions for this specific account."""
        statement = self.bank_app.get_account_transactions(self.account.acc_id)

        print(f"\n======================================")
        print(f"  FINAL STATEMENT: {self.account.name} (ID: {self.account.acc_id})"
        print(f"======================================")

        # Filter for only transactions performed during the current user session (o
        # Since we cannot easily track the "start" of the session without more comp
        # we will display all transactions associated with the account, newest firs

        account_transactions = self.bank_app.get_account_transactions(self.account.

        if not account_transactions:
            print("No transactions recorded.")
        else:
            for t in reversed(account_transactions):
                print(t)

        print("\n--------------------------------------")
        print(f"Current Final Balance: Rs.{self.account.balance:,.2f}")
        print("--------------------------------------\n")
```

Identify possible Exceptions and implement them

In [19]:

```
import datetime

# --- 1. Base Account Hierarchy (Refined for Rs. currency and method returns) ---
```

```python
class Account:
    """Superclass representing a generic bank account."""
    def __init__(self, acc_id, name, balance=0.0):
        self.acc_id = acc_id
        self.name = name
        self.balance = max(0.0, float(balance))

    def __str__(self):
        return f"ID: {self.acc_id}, Name: {self.name}, Balance: Rs.{self.balance:,.

    def deposit(self, amount):
        """Basic deposit: Adds funds and returns True on success."""
        amount = float(amount)
        if amount > 0:
            self.balance += amount
            return True
        return False

    def withdraw(self, amount):
        """Basic withdraw: Removes funds if available and returns True on success."
        amount = float(amount)
        if amount > 0 and self.balance >= amount:
            self.balance -= amount
            return True
        return False

class SavingsAccount(Account):
    """Subclass with specific rules for minimum balance and deposit limit."""
    MAX_DEPOSIT_LIMIT = 100000.00
    MIN_PERSONAL_BALANCE = 5000.00

    def __init__(self, acc_id, name, type, balance=0.0):
        valid_types = ['personal', 'corporate']
        self.type = type.lower() if type.lower() in valid_types else 'personal'
        super().__init__(acc_id, name, balance)

    def __str__(self):
        base_str = super().__str__()
        return f"{base_str}, Type: Savings ({self.type.capitalize()})"

    def deposit(self, amount):
        """Max Rs. 1 Lakh can be deposited at a time."""
        amount = float(amount)
        if amount > self.MAX_DEPOSIT_LIMIT:
            print(f"🛑 Error: Deposit limit exceeded. Max allowed is Rs.{self.MAX_I
            return False
        return super().deposit(amount)

    def withdraw(self, amount):
        """Min balance Rs. 5000 must be maintained (unless corporate, then balance
        amount = float(amount)
        remaining_balance = self.balance - amount

        if self.type == 'corporate' and remaining_balance >= 0:
            return super().withdraw(amount)
```

```
        elif self.type == 'personal' and remaining_balance >= self.MIN_PERSONAL_BAL
            return super().withdraw(amount)

        else:
            if self.type == 'personal':
                max_withdraw = max(0, self.balance - self.MIN_PERSONAL_BALANCE)
                print(f"🛑 Error: Personal accounts must maintain Rs.{self.MIN_PERS
            else:
                print("🛑 Error: Corporate account insufficient funds.")
            return False

class CurrentAccount(Account):
    """Subclass with specific rules for minimum balance and deposit limit."""
    MAX_DEPOSIT_LIMIT = 200000.00  # Rs. 2 Lakh
    MIN_BALANCE = 10000.00         # Rs. 10 Thousand

    def __init__(self, acc_id, name, balance=0.0):
        super().__init__(acc_id, name, balance)

    def __str__(self):
        base_str = super().__str__()
        return f"{base_str} | Type: Current"

    def deposit(self, amount):
        """Max Rs. 2 Lakh can be deposited at a time."""
        amount = float(amount)
        if amount > self.MAX_DEPOSIT_LIMIT:
            print(f"🛑 Error: Deposit limit exceeded. Max allowed is Rs.{self.MAX_I
            return False
        return super().deposit(amount)

    def withdraw(self, amount):
        """Min balance Rs. 10,000 must be maintained."""
        amount = float(amount)
        remaining_balance = self.balance - amount

        if remaining_balance >= self.MIN_BALANCE:
            return super().withdraw(amount)
        else:
            max_withdraw = max(0, self.balance - self.MIN_BALANCE)
            print(f"🛑 Error: Min balance of Rs.{self.MIN_BALANCE:,.2f} required. M
            return False

# --- 2. Transaction Class ---

class Transaction:
    """Stores details for a single account transaction."""
    def __init__(self, acc_id, type, amount, new_balance):
        self.acc_id = acc_id
        self.type = type
        self.amount = amount
        self.new_balance = new_balance
        self.timestamp = datetime.datetime.now()

    def __str__(self):
        return (f"[{self.timestamp.strftime('%Y-%m-%d %H:%M:%S')}] "
```

```
                f"Acc ID: {self.acc_id} | Type: {self.type} | "
                f"Amount: Rs.{self.amount:,.2f} | New Balance: Rs.{self.new_balance

# --- 3. BankApp Class (Central Manager) ---

class BankApp:
    """Manages a collection of accounts and handles transactions."""
    def __init__(self):
        self.accounts = {}
        self.transactions = []
        self._next_id = 1001

    def add_account(self, account):
        """Adds a new account to the manager."""
        self.accounts[account.acc_id] = account
        self._next_id += 1

    def get_next_id(self):
        return self._next_id

    def find_account(self, acc_id):
        """Utility to retrieve an account by ID."""
        return self.accounts.get(acc_id)

    def get_account_transactions(self, acc_id):
        """Filters transactions for a specific account ID."""
        return [t for t in self.transactions if t.acc_id == acc_id]

    def withdraw_from_account(self, account: Account, amount: float) -> float:
        """
        Executes withdrawal, logs transaction, and returns new balance.
        (Refactored to take amount as argument, not input)
        """
        # Call the account's specific withdraw method which handles rules and updat
        if account.withdraw(amount):
            self.transactions.append(Transaction(account.acc_id, "Withdrawal", amou
            return account.balance
        return account.balance

    def deposit_to_account(self, account: Account, amount: float) -> float:
        """
        Executes deposit, logs transaction, and returns new balance.
        (Refactored to take amount as argument, not input)
        """
        # Call the account's specific deposit method which handles rules and update
        if account.deposit(amount):
            self.transactions.append(Transaction(account.acc_id, "Deposit", amount,
            return account.balance
        return account.balance

# --- 4. User Class (New Requirement) ---

class User:
    """Provides a menu-driven interface for a single account holder."""
    def __init__(self, account: Account, bank_app: BankApp):
        self.account = account
```

```
        self.bank_app = bank_app

    def _show_current_balance(self):
        """Prints the current account details."""
        print(f"\n[Current Balance for {self.account.name} (ID: {self.account.acc_i

    # New helper method for robust input handling
    def _get_valid_amount(self, prompt):
        """Helper to safely get a positive float amount from the user."""
        while True:
            try:
                amount_str = input(prompt)
                amount = float(amount_str)
                if amount <= 0:
                    print("❌ Amount must be positive.")
                    continue
                return amount
            except ValueError:
                print("❌ Invalid input. Please enter a numerical amount.")
                # Loop continues until valid input is provided

    def run_menu(self):
        """
        Main loop for the user interface.
        Displays statement and final balance upon exiting.
        """
        print(f"\nWelcome, {self.account.name}! Accessing account ID: {self.account

        while True:
            self._show_current_balance()
            print("\n==========================")
            print("  User Transaction Menu")
            print("==========================")
            print("1. Deposit Money")
            print("2. Withdraw Money")
            print("3. Exit to Main Menu and View Statement")

            choice = input("Enter your choice (1-3): ")

            if choice == '1':
                self._handle_deposit()
            elif choice == '2':
                self._handle_withdraw()
            elif choice == '3':
                print("\n--- Exiting User Menu ---")
                self._display_final_statement()
                break
            else:
                print("❗ Invalid choice. Please select 1, 2, or 3.")

    def _handle_deposit(self):
        """Prompts for deposit amount and calls BankApp method."""
        amount = self._get_valid_amount("Enter amount to deposit (Rs.): ")
        if amount is None:
            return
```

```
        initial_balance = self.account.balance
        new_balance = self.bank_app.deposit_to_account(self.account, amount)

        # Check if balance actually changed (deposit was successful based on subcla
        if new_balance > initial_balance:
             print(f"💰 Deposit successful. New Balance: Rs.{new_balance:,.2f}")
        # Failure message is handled by the Account method's print statement

    def _handle_withdraw(self):
        """Prompts for withdrawal amount and calls BankApp method."""
        amount = self._get_valid_amount("Enter amount to withdraw (Rs.): ")
        if amount is None:
            return

        initial_balance = self.account.balance
        new_balance = self.bank_app.withdraw_from_account(self.account, amount)

        # Check if balance actually changed (withdrawal was successful based on sub
        if new_balance < initial_balance:
             print(f"💸 Withdrawal successful. New Balance: Rs.{new_balance:,.2f}")
        # Failure message is handled by the Account method's print statement

    def _display_final_statement(self):
        """Displays all transactions for this specific account."""
        statement = self.bank_app.get_account_transactions(self.account.acc_id)

        print(f"\n====================================")
        print(f"  FINAL STATEMENT: {self.account.name} (ID: {self.account.acc_id})")
        print(f"====================================")

        # Filter for only transactions performed during the current user session (o
        # Since we cannot easily track the "start" of the session without more comp
        # we will display all transactions associated with the account, newest firs

        account_transactions = self.bank_app.get_account_transactions(self.account.

        if not account_transactions:
            print("No transactions recorded.")
        else:
            for t in reversed(account_transactions):
                print(t)

        print("\n------------------------------------")
        print(f"Current Final Balance: Rs.{self.account.balance:,.2f}")
        print("------------------------------------\n")


# --- DEMONSTRATION MENU (Modified to use User class) ---

def main_menu():
    """Simple menu to test the BankApp functionalities."""
    bank = BankApp()

    # Initialize some accounts for testing
    bank.add_account(SavingsAccount(bank.get_next_id(), "Priya Sharma", "personal",
    bank.add_account(CurrentAccount(bank.get_next_id(), "Tech Solutions Pvt. Ltd.",
```

```
    bank.add_account(SavingsAccount(bank.get_next_id(), "Harsh Patel", "corporate",

    while True:
        print("\n======================================")
        print("  Gemini Bank System: Main Menu")
        print("======================================")
        print("1. Display All Accounts (Admin View)")
        print("2. Access Account (User Login)")
        print("3. View Global Transaction History")
        print("4. Exit Application")

        choice = input("Enter your choice: ")

        if choice == '1':
            print("\n--- All Active Accounts ---")
            if not bank.accounts:
                print("No accounts registered.")
                continue
            for acc in bank.accounts.values():
                print(acc)

        elif choice == '2':
            try:
                acc_id = int(input("Enter Account ID to access: "))
                account = bank.find_account(acc_id)

                if account:
                    # Launch the user interface for the selected account
                    user_interface = User(account, bank)
                    user_interface.run_menu()
                else:
                    print(f"🛑 Account ID {acc_id} not found.")
            except ValueError:
                print("❌ Invalid input. Please enter a numerical ID.")

        elif choice == '3':
            print("\n--- Global Transaction History ---")
            if not bank.transactions:
                print("No transactions recorded yet.")
                continue
            for t in reversed(bank.transactions): # Show newest first
                print(t)

        elif choice == '4':
            print("Thank you for using the Gemini Bank System. Goodbye!")
            break

        else:
            print("❗ Invalid choice. Please select from 1 to 4.")

if __name__ == '__main__':
    main_menu()
```

```
====================================
  Gemini Bank System: Main Menu
====================================
1. Display All Accounts (Admin View)
2. Access Account (User Login)
3. View Global Transaction History
4. Exit Application

--- All Active Accounts ---
ID: 1001, Name: Priya Sharma, Balance: Rs.50,000.00, Type: Savings (Personal)
ID: 1002, Name: Tech Solutions Pvt. Ltd., Balance: Rs.150,000.00 | Type: Current
ID: 1003, Name: Harsh Patel, Balance: Rs.75,000.00, Type: Savings (Corporate)

====================================
  Gemini Bank System: Main Menu
====================================
1. Display All Accounts (Admin View)
2. Access Account (User Login)
3. View Global Transaction History
4. Exit Application
❗ Invalid choice. Please select from 1 to 4.

====================================
  Gemini Bank System: Main Menu
====================================
1. Display All Accounts (Admin View)
2. Access Account (User Login)
3. View Global Transaction History
4. Exit Application
Thank you for using the Gemini Bank System. Goodbye!
```

Validate an IP Address

Difficulty: MediumAccuracy: 11.22%Submissions: 300K+Points: 4Average Time: 20m

You are given a string s in the form of an IPv4 Address. Your task is to validate an IPv4 Address, if it is valid return true otherwise return false.

> IPv4 addresses are canonically represented in dot-decimal notation, which consists of four decimal numbers, each ranging from 0 to 255, separated by dots, e.g., "172.16.254.1"
>
> A valid IPv4 Address is of the form x1.x2.x3.x4 where 0 <= (x1, x2, x3, x4) <= 255. Thus, we can write the generalized form of an IPv4 address as (0-255).(0-255).(0-255).(0-255)

Note: Here we are considering numbers only from 0 to 255 and any additional leading zeroes will be considered invalid.

Examples :

Input: s = "222.111.111.111" Output: true Explanation: Here, the IPv4 address is as per the criteria mentioned and also all four decimal numbers lies in the mentioned range.

Input: s = "5555..555" Output: false Explanation: "5555..555" is not a valid IPv4 address, as the middle two portions are missing.

Input: s = "0.0.0.255" Output: true

In [ ]: