

College of Science and Technology

Phuentsholing, Rinchending

TEMPA GYELTSHEN

02240134

Electronic and Communication

Part A:

"""

Enhanced Bank Application with Modern GUI

This version improves the visual design and user experience of the banking application

while maintaining all original functionality. Features include:

- Modern color scheme and styling

- Better widget organization

- Improved input validation

- Responsive layout

- Enhanced visual feedback

"""

import random

import os

import tkinter as tk

from tkinter import ttk, messagebox

# --- Custom Exceptions ---

class InputError(Exception):      #Exception raised for invalid user inputs.

    pass

class FundTransferError(Exception):

    pass

# --- Account Classes ---

class Account:                         #Base class for all account types.

    def \_\_init\_\_(self, number, pin, acc\_type, amount=0.0):

        self.num = number

        self.pin = pin

        self.type = acc\_type

        self.balance = amount

    def deposit(self, amount):      #Adds the specified amount to the account balance.

        if amount <= 0:

            raise InputError("Amount must be more than zero.")

        self.balance += amount

    def withdraw(self, amount):        #Deducts the specified amount from the balance if sufficient funds exist.

        if amount > self.balance:

            raise InputError("Not enough balance.")

        self.balance -= amount

    def transfer(self, amount, receiver): #Transfers funds to another account.

        if amount > self.balance:

            raise FundTransferError("Not enough funds to send.")

        self.withdraw(amount)

        receiver.deposit(amount)

    def recharge(self, number, amount):         #Simulates a mobile recharge by deducting the amount.

        if not number.isdigit() or len(number) != 10:

            raise InputError("Phone number must be 10 digits.")

        if amount > self.balance:

            raise InputError("Not enough balance for top-up.")

        self.balance -= amount

class Personal(Account):                    #Represents a personal account.

    def \_\_init\_\_(self, number, pin, balance=0.0):

        super().\_\_init\_\_(number, pin, "personal", balance)

class Business(Account):                   #Represents a business account.

    def \_\_init\_\_(self, number, pin, balance=0.0):

        super().\_\_init\_\_(number, pin, "business", balance)

# --- Core Banking System Logic ---

class BankCore:              #Manages all user accounts and data persistence.

    def \_\_init\_\_(self):

        self.users = {}

        self.\_load\_data()

    def \_load\_data(self):       #Loads account data from the local text file.

        if not os.path.isfile("data.txt"):

            return

        with open("data.txt", "r") as file:

            for line in file:

                acc, pwd, typ, bal = line.strip().split(",")

                cls = Personal if typ == "personal" else Business

                self.users[acc] = cls(acc, pwd, float(bal))

    def \_save\_data(self):               #Writes all current account data to the local text file.

        with open("data.txt", "w") as file:

            for acc in self.users.values():

                file.write(f"{acc.num},{acc.pin},{acc.type},{acc.balance}\n")

    def new\_account(self, kind):        #Creates and registers a new account.

        acc\_num = str(random.randint(10000, 99999))

        pin = str(random.randint(1000, 9999))

        user = Personal(acc\_num, pin) if kind == "personal" else Business(acc\_num, pin)

        self.users[acc\_num] = user

        self.\_save\_data()

        return acc\_num, pin

    def authenticate(self, acc\_num, pin):

#Authenticates a user based on account number and PIN.

        acc = self.users.get(acc\_num)

        if acc and acc.pin == pin:

            return acc

        raise InputError("Incorrect login details.")

    def remove\_account(self, acc\_num):                            #Deletes an existing account from the system.

        if acc\_num in self.users:

            del self.users[acc\_num]

            self.\_save\_data()

        else:

            raise InputError("Account not found.")

# --- Enhanced Graphical User Interface ---

class BankApp:                   #Modernized GUI for the banking application.

    def \_\_init\_\_(self, system):

        self.bank = system

        self.user = None

        self.app = tk.Tk()

        self.app.title("AchoDaka Bank")

        self.app.geometry("400x500")

        self.app.resizable(False, False)

        # Configure styles

        self.style = ttk.Style()

        self.style.configure('TFrame', background='#f0f0f0')

        self.style.configure('TLabel', background='#f0f0f0', font=('Times New Roman', 10))

        self.style.configure('Header.TLabel', font=('Times New Roman', 14, 'bold'))

        self.style.configure('TButton', font=('Times New Roman', 10), padding=5)

        self.style.configure('Primary.TButton', foreground='black', background='#0078d7')

        self.style.configure('Success.TButton', foreground='black', background='#4CAF50')

        self.style.configure('Danger.TButton', foreground='black', background='#f44336')

        # Main container

        self.main\_frame = ttk.Frame(self.app)

        self.main\_frame.pack(fill=tk.BOTH, expand=True, padx=20, pady=20)

        self.\_show\_login()

        self.app.mainloop()

    def \_clear(self):                  #Clears the current window content.

        for widget in self.main\_frame.winfo\_children():

            widget.destroy()

    def \_show\_login(self):    #Displays the login screen with modern styling.

        self.\_clear()

        # Header

        ttk.Label(self.main\_frame, text="AchoDaka Bank", style='Header.TLabel').pack(pady=(0, 20))

        # Login container

        form\_frame = ttk.Frame(self.main\_frame)

        form\_frame.pack(pady=10)

        # Account number field

        ttk.Label(form\_frame, text="Account Number:").grid(row=0, column=0, sticky=tk.W, pady=5)

        self.entry\_acc = ttk.Entry(form\_frame, font=('Times New Roman', 10))

        self.entry\_acc.grid(row=0, column=1, pady=5, padx=5)

        # PIN field

        ttk.Label(form\_frame, text="PIN:").grid(row=1, column=0, sticky=tk.W, pady=5)

        self.entry\_pin = ttk.Entry(form\_frame, show="•", font=('Times New Roman', 10))

        self.entry\_pin.grid(row=1, column=1, pady=5, padx=5)

        # Buttons container

        b\_layout = ttk.Frame(self.main\_frame)

        b\_layout.pack(pady=20)

        # Login

        ttk.Button(b\_layout, text="Login", command=self.\_login, style='Primary.TButton').pack(fill=tk.X, pady=5)

        # Registration buttons

        ttk.Label(b\_layout, text="Don't have an account?").pack(pady=(10, 5))

        ttk.Button(b\_layout, text="Open Personal Account",

                  command=lambda: self.\_register("personal"), style='Success.TButton').pack(fill=tk.X, pady=5)

        ttk.Button(b\_layout, text="Open Business Account",

                  command=lambda: self.\_register("business"), style='Success.TButton').pack(fill=tk.X, pady=5)

        # Focus on account number field

        self.entry\_acc.focus\_set()

    def \_register(self, acc\_type):          #Handles new account registration with visual feedback.

        acc\_num, pin = self.bank.new\_account(acc\_type)

        messagebox.showinfo("Account Created",

                          f"Your new {acc\_type} account has been created!\n\n"

                          f"Account Number: {acc\_num}\n"

                          f"PIN: {pin}\n\n"

                          "Please keep this information secure.")

    def \_login(self):           #Attempts to log the user into the system.

        num = self.entry\_acc.get()

        pin = self.entry\_pin.get()

        try:

            self.user = self.bank.authenticate(num, pin)

            self.dashboard()

        except InputError as er:

            messagebox.showerror("Login Fails", str(er))

            self.entry\_pin.delete(0, tk.END)

            self.entry\_pin.focus\_set()

    def dashboard(self):             #Displays the main dashboard with account controls.

        self.\_clear()

        # Header with user info

        ttk.Label(self.main\_frame,

                 text=f"Welcome, {self.user.type.capitalize()} User\nAccount: {self.user.num}",

                 style='Header.TLabel').pack(pady=(0, 20))

        # Balance display

        amount\_layout = ttk.Frame(self.main\_frame)

        amount\_layout.pack(fill=tk.X, pady=10)

        ttk.Label(amount\_layout, text="Balance:").pack(side=tk.LEFT)

        ttk.Label(amount\_layout, text=f"Nu. {self.user.balance:.1f}",

                 font=('Times New Roman', 12, 'bold')).pack(side=tk.RIGHT)

        # Action buttons

        b\_layout = ttk.Frame(self.main\_frame)

        b\_layout.pack(fill=tk.BOTH, expand=True)

        ttk.Button(b\_layout, text="Deposit", command=self.\_deposit,

                  style='Primary.TButton').pack(fill=tk.X, pady=5)

        ttk.Button(b\_layout, text="Withdraw", command=self.\_withdraw,

                  style='Primary.TButton').pack(fill=tk.X, pady=5)

        ttk.Button(b\_layout, text="Transfer Money", command=self.\_transfer,

                  style='Primary.TButton').pack(fill=tk.X, pady=5)

        ttk.Button(b\_layout, text="Mobile Recharge", command=self.\_recharge,

                  style='Primary.TButton').pack(fill=tk.X, pady=5)

        ttk.Button(b\_layout, text="View Balance", command=self.\_balance,

                  style='Success.TButton').pack(fill=tk.X, pady=5)

        ttk.Button(b\_layout, text="Delete Account", command=self.\_delete\_account,

                  style='Danger.TButton').pack(fill=tk.X, pady=5)

        ttk.Button(b\_layout, text="Logout", command=self.\_logout).pack(fill=tk.X, pady=(20, 5))

    def \_balance(self):              #Shows the current account balance in a styled dialog.

        messagebox.showinfo("Account Balance",

                          f"Your current balance is:\n\nNu. {self.user.balance:.2f}")

    def \_deposit(self):           #Handles deposit transactions with input validation.

        amt = self.\_prompt\_amount("Enter deposit amount:")

        if amt:

            try:

                self.user.deposit(amt)

                self.bank.\_save\_data()

                messagebox.showinfo("Success", f"Deposit of Nu. {amt:.2f} was successful.\nNew balance: Nu. {self.user.balance:.2f}")

            except InputError as e:

                messagebox.showerror("Deposit Failed", str(e))

    def \_withdraw(self):         #Handles withdrawals with confirmation.

        amt = self.\_prompt\_amount("Enter withdrawal amount:")

        if amt:

            try:

                self.user.withdraw(amt)

                self.bank.\_save\_data()

                messagebox.showinfo("Success", f"Withdrawal of Nu. {amt:.2f} was successful.\nNew balance: Nu. {self.user.balance:.2f}")

            except InputError as e:

                messagebox.showerror("Withdrawal Failed", str(e))

    def \_transfer(self):        #Handles money transfers between accounts.

        receiver = self.\_prompt("Enter receiver's account number:")

        if not receiver:

            return

        if receiver == self.user.num:

            messagebox.showerror("Error", "Cannot transfer to your own account")

            return

        amt = self.\_prompt\_amount("Enter amount to transfer:")

        if amt:

            try:

                target = self.bank.users.get(receiver)

                if not target:

                    raise FundTransferError("Receiver account not found.")

                if messagebox.askyesno("Confirm Transfer",

                                     f"Transfer Nu. {amt:.2f} to account {receiver}?"):

                    self.user.send\_money(amt, target)

                    self.bank.\_save\_data()

                    messagebox.showinfo("Success",

                                      f"Transfer of Nu. {amt:.2f} to account {receiver} was successful.\n"

                                      f"New balance: Nu. {self.user.balance:.2f}")

            except (InputError, FundTransferError) as e:

                messagebox.showerror("Transfer Failed", str(e))

    def \_recharge(self):           #Handles mobile recharge with validation.

        number = self.\_prompt("Enter 10-digit mobile number:")

        if not number:

            return

        amt = self.\_prompt\_amount("Enter recharge amount:")

        if amt:

            try:

                if messagebox.askyesno("Confirm Recharge",

                                     f"Recharge Nu. {amt:.2f} to number {number}?"):

                    self.user.recharge(number, amt)

                    self.bank.\_save\_data()

                    messagebox.showinfo("Success",

                                      f"Recharge of Nu. {amt:.2f} to {number} was successful.\n"

                                      f"New balance: Nu. {self.user.balance:.2f}")

            except InputError as e:

                messagebox.showerror("Recharge Failed", str(e))

    def \_delete\_account(self):   #Handles account deletion with confirmation.

        if messagebox.askyesno("Confirm delete action",

                             "Are you sure?\n"

                             "This action is permanent."):

            try:

                self.bank.remove\_account(self.user.num)

                messagebox.showinfo("Account successfully deleted.")

                self.\_show\_login()

            except InputError as e:

                messagebox.showerror("Account deletion Fails", str(e))

    def \_logout(self):          #returns to login screen.

        self.user = None

        self.\_show\_login()

    def \_prompt(self, message):

        """Creates a custom styled input dialog."""

        dialog = tk.Toplevel(self.app)

        dialog.title("Input Required")

        dialog.resizable(False, False)

        ttk.Label(dialog, text=message).pack(pady=10, padx=20)

        entry = ttk.Entry(dialog, font=('Helvetica', 10))

        entry.pack(pady=5, padx=20)

        result = []

        def submit():

            result.append(entry.get())

            dialog.destroy()

        b\_layout = ttk.Frame(dialog)

        b\_layout.pack(pady=10)

        ttk.Button(b\_layout, text="OK", command=submit, style='Primary.TButton').pack(side=tk.LEFT, padx=5)

        ttk.Button(b\_layout, text="Cancel", command=dialog.destroy).pack(side=tk.LEFT, padx=5)

        entry.focus\_set()

        dialog.grab\_set()

        dialog.wait\_window()

        return result[0] if result else None

    def \_prompt\_amount(self, message):  #Prompts for a numeric value with validation.

        try:

            value = self.\_prompt(message)

            if not value:

                return None

            amount = float(value)

            if amount <= 0:

                raise ValueError("Invalid Amount")

            return amount

        except ValueError:

            messagebox.showerror("Please enter a valid amount.")

            return None

# --- Application Launch ---

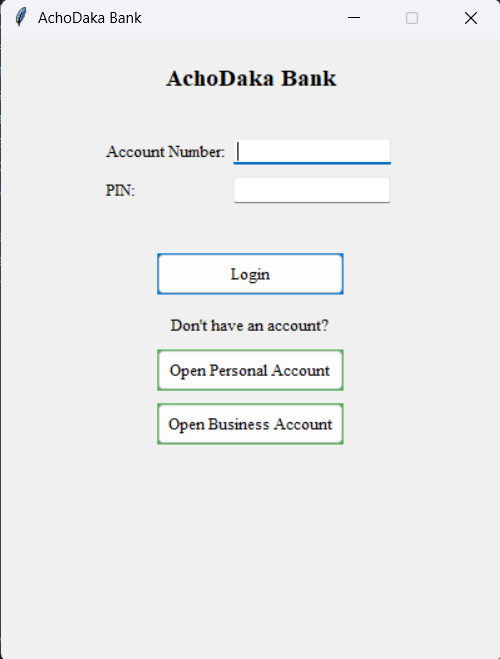
if \_\_name\_\_ == "\_\_main\_\_":

    system = BankCore()

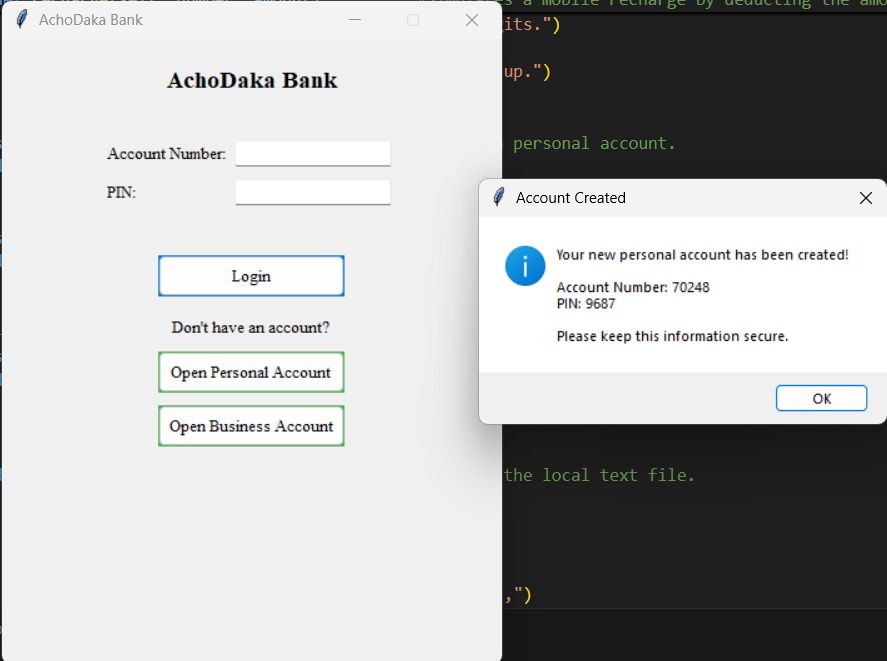
    BankApp(system)

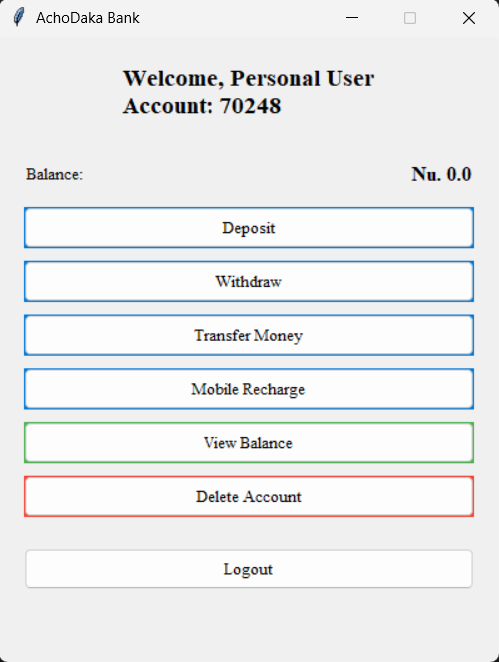
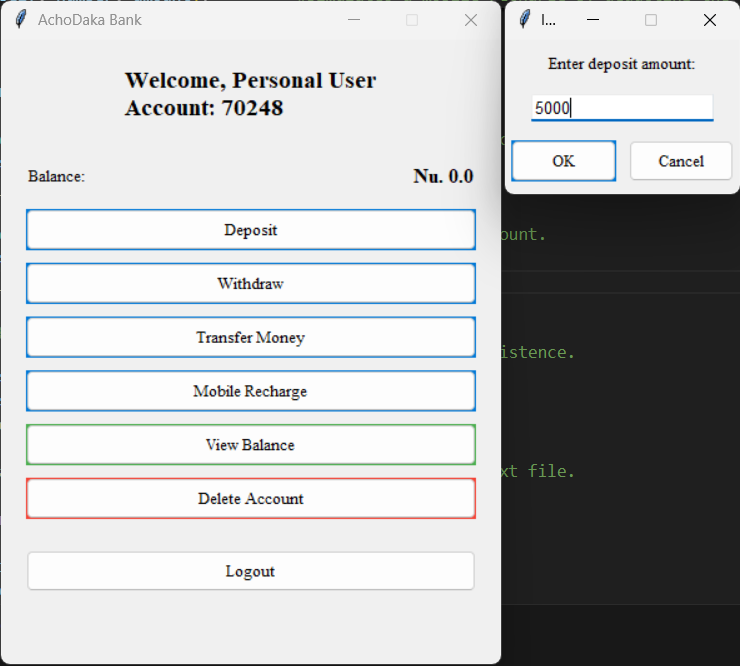
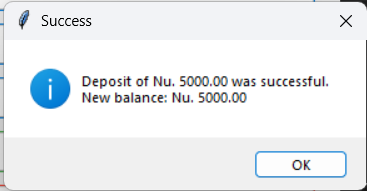
Outputs:

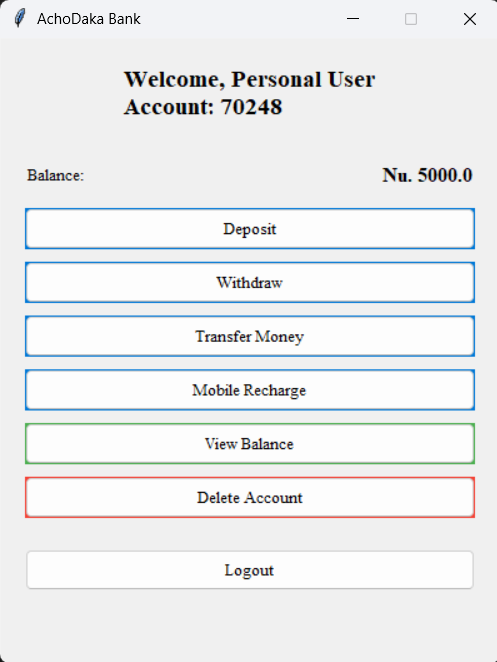
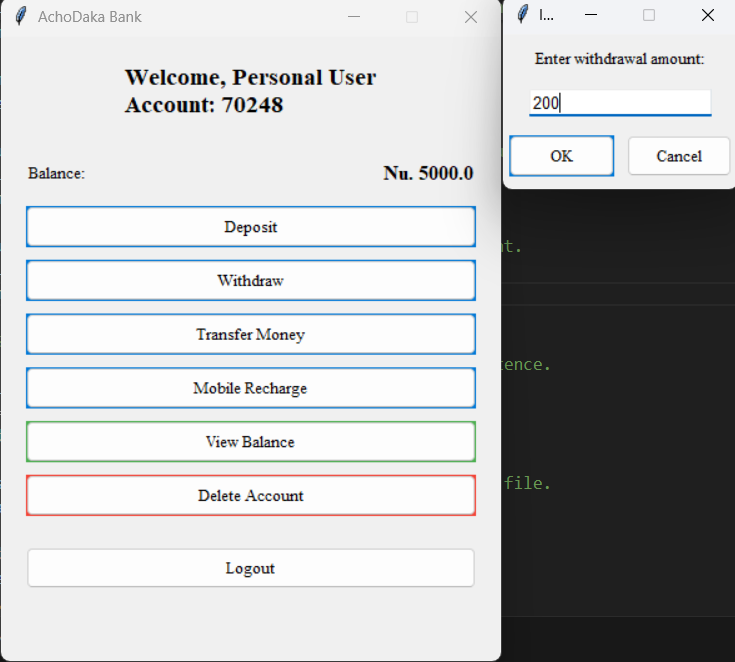
1. Login Page



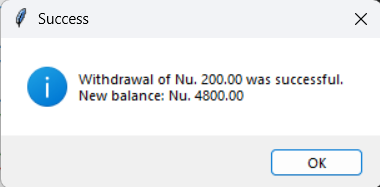
Login Page for AchoDaka Banking system

1. Open Personal Account

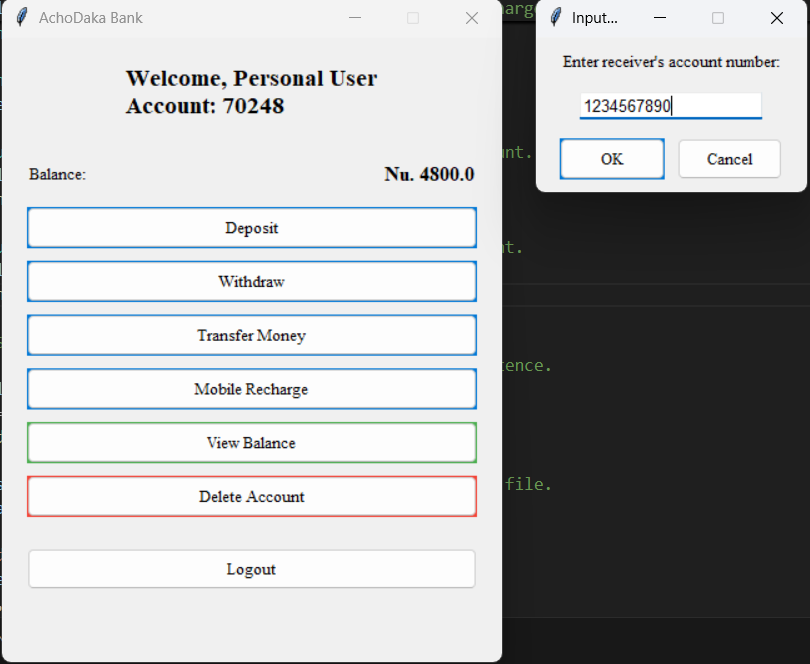
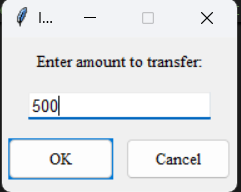
Deposit features for Personal account

Withdrawing features

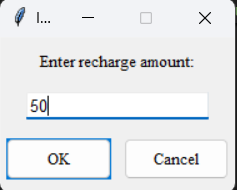
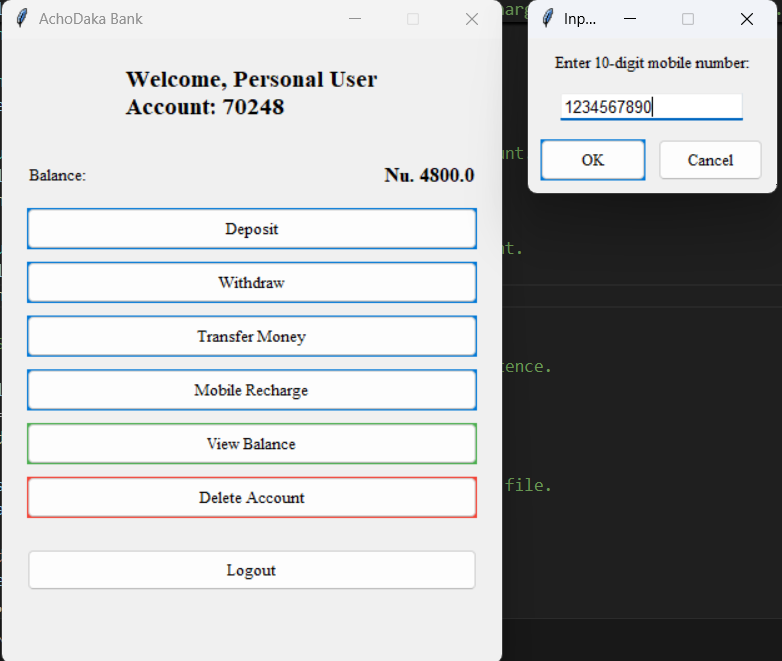
The Amounts get updated in Balance dashboard

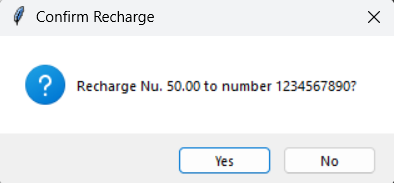


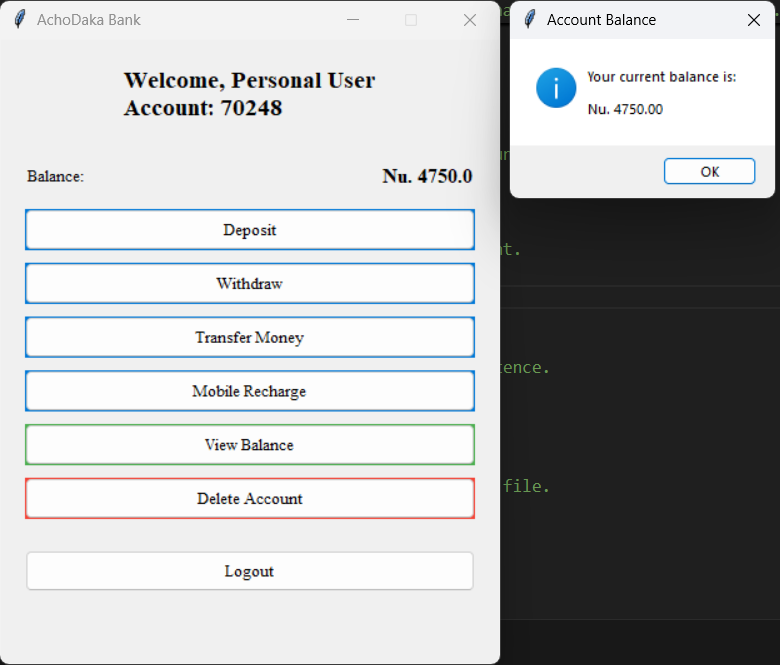
Textbox displaying success message

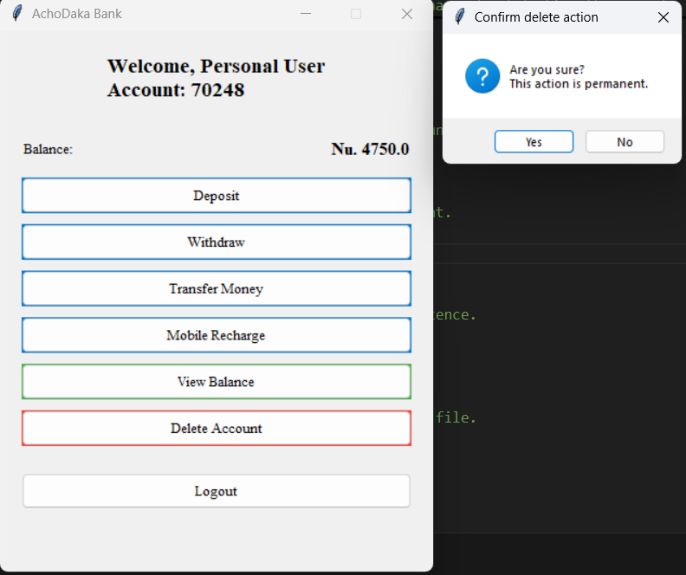
Fund Transfer features

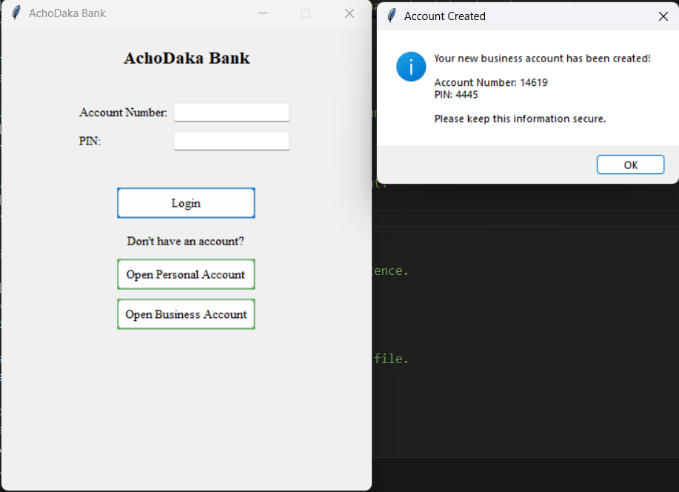
Top-Up features

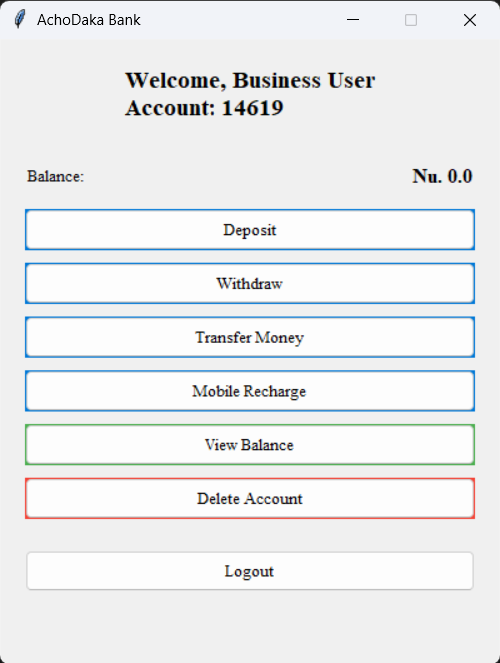




Viewing balance:

Account deletion process

1. Business Account Creation

Business account features

Business account holds similar feature as Personal account as displayed above.

Part B

"""

Unit Test Suite for Simple Banking Application

This module uses Python's unittest framework to validate core banking features.

It covers tests for user inputs, error handling, and functionality for deposits,

withdrawals, transfers, recharge, and authentication.

"""

import unittest

from TempaGyeltshen\_02240134\_A3\_PA import (

    Personal, Business, BankCore, InputError, FundTransferError

)

class SimpleBankTest(unittest.TestCase):

    """

    Test class to evaluate individual banking features and edge cases.

    Includes tests for valid and invalid scenarios across core functionalities.

    """

    def setUp(self):                         #Initializes test accounts before each test.

        self.bank = BankCore()

        self.acc1 = Personal("11111", "1111", 1000)  # Account with initial Nu. 1000

        self.acc2 = Business("22222", "2222", 500)   # Account with initial Nu. 500

        self.bank.user["11111"] = self.acc1

        self.bank.user["22222"] = self.acc2

    # --- Unusual Inputs ---

    def test\_negative\_deposit(self):             #Check deposit with negative amount raises InputError.

        with self.assertRaises(InputError):

            self.acc1.deposit(-100)  # Invalid: can't deposit negative value

    def test\_invalid\_recharge\_number(self):                   #Ensure recharge with letters in phone number is rejected.

        with self.assertRaises(InputError):

            self.acc1.recharge("abc123", 50)

    def test\_short\_number(self):                              #Ensure too-short phone number is rejected.

        with self.assertRaises(InputError):

            self.acc1.recharge("123", 50)

    # --- Invalid Usage Cases ---

    def test\_withdraw\_too\_much(self):                            #Withdraw more than balance should raise InputError.

        with self.assertRaises(InputError):

            self.acc2.withdraw(10000)

    def test\_transfer\_too\_much(self):                          #Attempting to transfer more than available balance should raise error.

        with self.assertRaises(FundTransferError):

            self.acc1.transfer(2000, self.acc2)

    def test\_recharge\_over\_balance(self):                        #Mobile recharge amount exceeding balance should fail.

        with self.assertRaises(InputError):

            self.acc2.recharge("1234567890", 10000)

    # --- Valid Operations ---

    def test\_deposit(self):                                #Successful deposit should increase balance.

        self.acc1.deposit(500)

        self.assertEqual(self.acc1.balance, 1500)

    def test\_withdraw(self):                               #Successful withdrawal should decrease balance.

        self.acc2.withdraw(300)

        self.assertEqual(self.acc2.balance, 200)

    def test\_transfer(self):                               #Valid fund transfer should update both balances correctly."""

        self.acc1.transfer(200, self.acc2)

        self.assertEqual(self.acc1.balance, 800)

        self.assertEqual(self.acc2.balance, 700)

    def test\_valid\_recharge(self):                            #Valid mobile recharge should deduct amount from account."""

        self.acc1.recharge("1234567890", 100)

        self.assertEqual(self.acc1.balance, 900)

    def test\_remove\_account(self):                            #Account removal should delete user from records."""

        self.bank.remove\_account("11111")

        self.assertNotIn("11111", self.bank.user)

    def test\_login\_success(self):                              #Successful login returns the correct account object."""

        user = self.bank.authenticate("22222", "2222")

        self.assertEqual(user, self.acc2)

    def test\_login\_fail(self):                                 #Login with wrong PIN should raise InputError."""

        with self.assertRaises(InputError):

            self.bank.authenticate("22222", "0000")

# --- Execute all test cases ---

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

Output:

