Project

Bank Machine

Using Java and Object-Oriented Programming Techniques

Dildeep Dhillon
Fall 2023

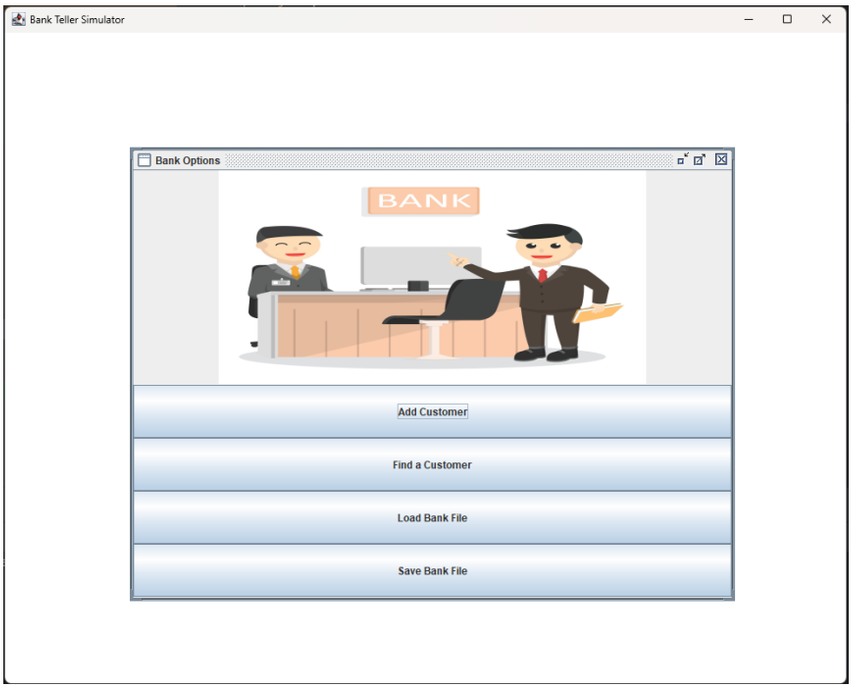
Introduction:

Used Java in combination with object-oriented programming methods like inheritance and polymorphism to create a bank machine that allows a user to create accounts, add additional accounts, and manage all accounts by depositing, withdrawing, or checking the balance of these accounts. Additionally, Java swing libraries were used to create a user-friendly interface, to interact with a customer's accounts to perform these actions. The program also allows the user to save or load the program from where it last left off.

The application is designed to store a customer's information, such as his contact information, what bank he is using, and how many accounts he has open with this bank. Also, each account will hold a balance, and a list of transactions that have taken place. A bank teller can use it to pull up the necessary information for a customer, and be able to make withdraws, deposits, close or open accounts on the customers behalf. This project is of interest to me, because databases are needed almost everywhere, and banking is something everyone interacts with at least once a month. So, this is application would be a small reflection, of what your bank teller could be using when processing your requests.

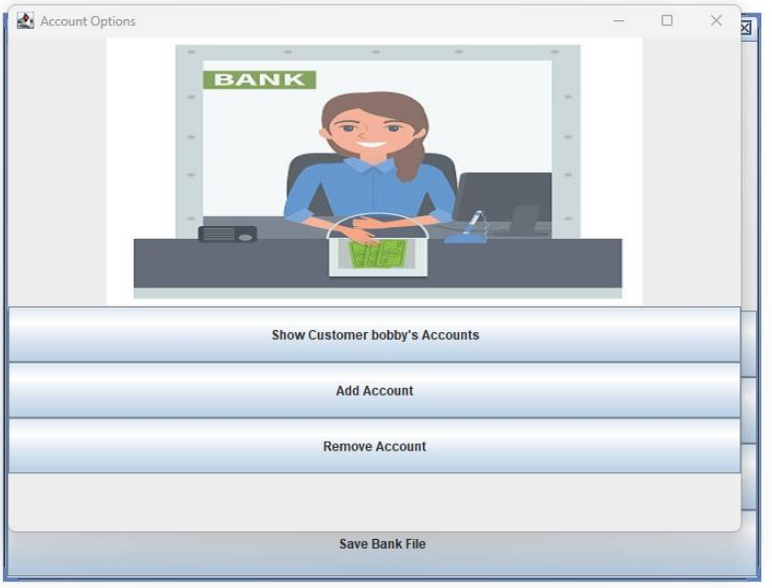
Snapshots of Program in Action:

Main Menu:

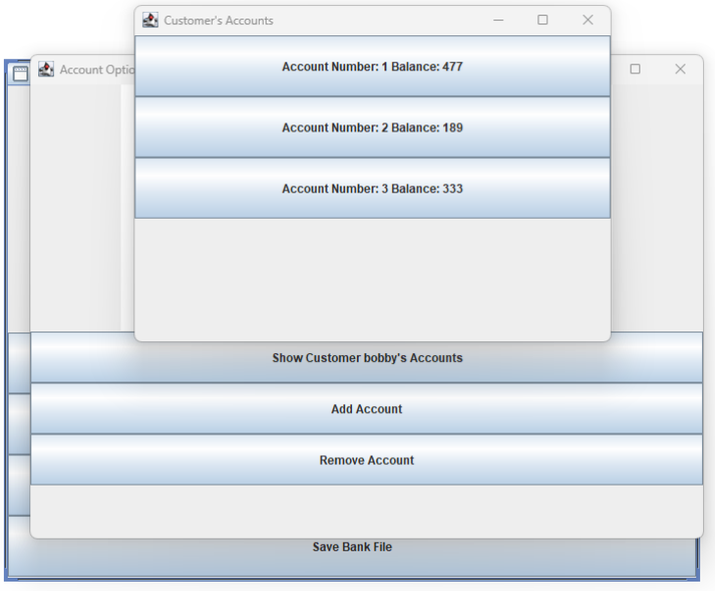


Add Customer:

Customer’s Main Menu (Bobby in this case):

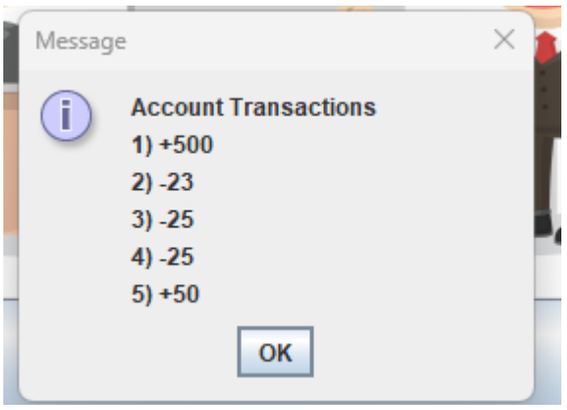


Customer’s Accounts:

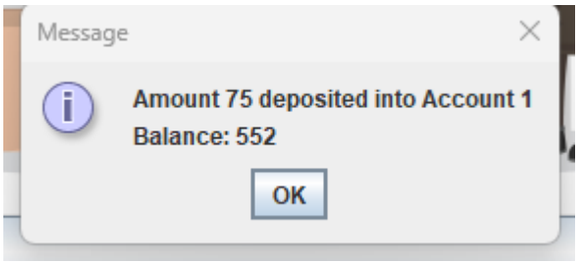


Selected Accounts Menu (Account 1 in this case):

Show Account Transactions:



Deposit Money in Account ($75 in this case):



There are many more menu options that can be shown per request.

Some Code Snippets (more available on request):

Account Class:

```
public class Account implements Writable {
    private int balance;
    private final int accountNumber;
    private final List<String> transactions;

    // REQUIRES: accountNumber > 0
    // EFFECTS: constructs the account with the balance set to $0 and
the given account number
    // and creates a list for keeping track of transactions
    public Account(int accountNumber) {
        this.accountNumber = accountNumber;
        this.balance = 0;
        this.transactions = new ArrayList<>();
    }

    // Overloaded constructor, so we have the option to create an
account with a balance of zero
    // or give an account an initial balance
    // REQUIRES: accountNumber > 0 and initialBalance > 0
    // EFFECTS: constructs the account with the balance set to the
given amount and the given account number
    // and creates a list for keeping track of transactions
    public Account(int accountNumber, int initialBalance) {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
        this.transactions = new ArrayList<>();
    }

    // REQUIRES: value > 0
    // MODIFIES: this
    // EFFECTS: adds the value to the balance, and updates the
transactions list
```

```
    public void deposit(int value) {
        this.balance += value;
        this.transactions.add("+" + value);
        EventLog.getInstance().logEvent(new Event("Deposited " + value
+ " to Account "
                + accountNumber + "."));
    }

    // REQUIRES: value > 0
    // MODIFIES: this
    // EFFECTS: if there is sufficient balance on the account
    //              - subtracts the value from the balance
    //              - updates the transactions list
    //              - returns true
    //              otherwise, return false
    public boolean withdraw(int value) {
        if (this.balance >= value) {
            this.balance -= value;
            this.transactions.add("-" + value);
            EventLog.getInstance().logEvent(new Event("Withdrew " +
value + " from Account "
                    + accountNumber + "."));
            return true;
        } else {
            return false;
        }

    }
  // More code than shown
}
```

Bank Class:

```
public class Bank implements Writable {
    private String name;
    private final List<Customer> customers;


    // EFFECTS: constructs the bank, by creating the list to store the
customers
    //          and with the given name
    public Bank(String name) {
        this.name = name;
        customers = new ArrayList<>();
    }

    // Overloaded constructor, so we have the option to create a bank
without a name
    // EFFECTS: constructs the bank, by creating the list to store the
customers
    public Bank() {
        this.name = null;
        customers = new ArrayList<>();
    }

    // REQUIRES: customer1 != null
    // MODIFIES: this
    // EFFECTS: adds a customer to the banks list of customers
    public void addACustomer(Customer customer1) {
        customers.add(customer1);
        EventLog.getInstance().logEvent(new Event("Customer " +
customer1.getName() + " added."));
    }

    // REQUIRES: name & phone != null
    // EFFECTS: if the name and phone number match a customer in the
customers list
    //          - return the customer
    //          - otherwise, return null
    public Customer findACustomer(String name, String phone) {
        for (Customer customer1 : customers) {
```

```
            if (customer1.getName().equals(name) &&
customer1.getPhone().equals(phone)) {
                EventLog.getInstance().logEvent(new Event("Customer "
+ customer1.getName() + " found."));
                return customer1;
            }
        }
        return null;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public List<Customer> getCustomers() {
        return this.customers;
    }

        // More code than shown
}
```

Customer Class:

```
public class Customer implements Writable {
    private final String name;
    private String phone;
    private final List<Account> accounts;

    // REQUIRES: name and phone number != null
    // EFFECTS: constructs the bank Customer, with given name and
phone number
    //          also creates a list of bank accounts for the customer
    public Customer(String name, String phone) {
        this.name = name;
        this.phone = phone;
        accounts = new ArrayList<>();
    }

    // REQUIRES: account != null
    // MODIFIES: this
    // EFFECTS: adds a bank account to a customers list of bank
accounts
    public void addAccount(Account account) {
        accounts.add(account);
        EventLog.getInstance().logEvent(new Event("Account " +
account.getAccountNumber() + " added."));
    }


    // EFFECTS: checks what the largest account number in the list of
account numbers is
    //          and return that accountNumber + 1
    public int getNextAccountNumber() {
        int nextNumber = 0;
        for (Account account : accounts) {
            if (account.getAccountNumber() > nextNumber) {
                nextNumber = account.getAccountNumber();
            }
        }
        return nextNumber + 1;
    }

    // REQUIRES: accountNumber > 0
    // MODIFIES: this
    // EFFECTS: if the given account number exists and the balance of
the account is zero
    //          - removes a bank account with the given account number
```

```
from the customers list of accounts
    //          - return true
    //          - otherwise, false
    public boolean removeAccount(int accountNumber) {
        for (Account account : accounts) {
            if (account.getBalance() == 0 &&
account.getAccountNumber() == accountNumber) {
                EventLog.getInstance().logEvent(new Event("Account " +
account.getAccountNumber()
                        + " removed."));
                accounts.remove(account);
                return true;
            }
        }
        return false;
    }

    // REQUIRES: accountNumber > 0
    // MODIFIES: this
    // EFFECTS: if the given account number exists in the accounts
list
    //          - return that account
    //          - otherwise, return null
    public Account getSingleAccount(int accNumber) {
        for (Account account : accounts) {
            if (account.getAccountNumber() == accNumber) {
                return account;
            }
        }
        return null;
    }

    // REQUIRES: phone != null
    // MODIFIES: this
    // EFFECTS: sets the phone to the given value
    public void setPhone(String phone) {
        this.phone = phone;
    }

    public String getName() {
        return this.name;
    }

    public String getPhone() {
        return this.phone;
    }

    public List<Account> getAccounts() {
        return this.accounts;
    }

   // More code than shown
}
```

BankTellerUI Class:

```
public class BankTellerUI extends JFrame {
    private static final int WIDTH = 1000;
    private static final int HEIGHT = 800;
    private Bank bank;
    JDesktopPane desktop = new JDesktopPane();
    private final JInternalFrame controlPanel;
    private Customer currentCustomer;
    private JTextField customerName;
    private JTextField customerPhone;
    private Account currentAccount;
    private static final String JSON_STORE = "./data/myFile.json";
    private final JsonWriter jsonWriter = new JsonWriter(JSON_STORE);
    private final JsonReader jsonReader = new JsonReader(JSON_STORE);

    // Overall structure references
```

```
https://github.students.cs.ubc.ca/CPSC210/AlarmSystem
    // MODIFIES: this
    // EFFECTS: initializes the bank object, creates a JDesktopPane
and adds a mouse listener to it. Creates a
    //          JInternalFrame and positions and resizes it. Runs the
addButtonMainPanel() method. Adds the
    //          JInternalFrame to the JDesktopPane.
    public BankTellerUI() {
        bank = new Bank();
        desktop.addMouseListener(new DesktopFocusAction());
        controlPanel = new JInternalFrame("Bank Options", true, true,
true, true);
        controlPanel.setLayout(new BorderLayout());

        addButtonMainPanel();

        // when the control panel is closing, it will run the
printLogEvents() method.
        controlPanel.addInternalFrameListener(new
InternalFrameAdapter() {
            @Override
            public void internalFrameClosing(InternalFrameEvent e) {
                printLogEvents();
            }
        });

        setContentPane(desktop);
        setTitle("Bank Teller Simulator");
        setSize(WIDTH, HEIGHT);

        controlPanel.pack();
        controlPanel.setVisible(true);

        int panelWidth = (WIDTH - controlPanel.getWidth()) / 2;
        int panelHeight = (HEIGHT - controlPanel.getHeight()) / 2;

        controlPanel.setLocation(panelWidth, panelHeight);
        desktop.add(controlPanel);

        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        centerOnScreen();
        setVisible(true);

    }


    // MODIFIES: this
    // EFFECTS: centers JDesktopPane on desktop window
    private void centerOnScreen() {
        int width = Toolkit.getDefaultToolkit().getScreenSize().width;
        int height =
Toolkit.getDefaultToolkit().getScreenSize().height;
        setLocation((width - getWidth()) / 2, (height - getHeight()) /
2);
    }

  // A lot more code in this class, to help with building UI

}
```

```
Stores the saved data:
{
  "Customers": [
    {
      "name": "bobby",
      "phone number": "6049991111",
      "Accounts": [
        {
          "Account Number": 1,
          "balance": 477,
          "transactions": [
```

```
            "+500",
            "-23",
            "-25",
            "-25",
            "+50"
          ]
        },
        {
          "Account Number": 2,
          "balance": 189,
          "transactions": [
            "-250",
            "+50",
            "-111"
          ]
        },
        {
          "Account Number": 3,
          "balance": 333,
          "transactions": []
        }
      ]
    },
    {
      "name": "ben",
      "phone number": "6041119999",
      "Accounts": [
        {
          "Account Number": 1,
          "balance": 80,
          "transactions": [
            "+333",
            "-233",
            "-20"
          ]
        },
        {
          "Account Number": 2,
          "balance": 300,
          "transactions": []
        },
        {
          "Account Number": 3,
          "balance": 109,
          "transactions": [
            "+159",
            "-50"
          ]
        }
      ]
    },
    {
      "name": "bob",
      "phone number": "6041119999",
      "Accounts": [{
        "Account Number": 1,
        "balance": 197,
        "transactions": [
          "-50",
          "+25"
        ]
      }]
    }
  ],
  "Bank name": "New bank"
}
```

UML Design Diagram: