****Tarlac State University

**COLLEGE OF COMPUTER STUDIES**

Case Study

in

Computer Programming 2

Submitted By:

**Manalo, Marc Rodnie**

**Romero, Jordan Pierce**

**Yango, Sean Stephen Rupert**

Submitted To:

**Asidera, Rovell**

Date: **May 17, 2023**

In conclusion po, we developed an ATM Banking system, where our program works like an actual ATM with its various functions po. The goal po nung program namin is whether bago ka palang po sa programming or may experience na is maiintindian po agad nila ito. Our program provides simple interface that is easy to use and codes that is easy to understand. Our program can serve as an educational tool for its concepts such as inheritance, Java Swing, and many more. This ATM banking system offers a practical and accessible way to learn and apply programming concepts.

**INTRODUCTION**

In this case study, we will discuss the development of an ATM Banking System using Java. The main objective is to create a simulated banking system that allows users to perform common banking tasks. The system includes features such as managing accounts, transferring funds, checking balances, and processing transactions. The goal of the ATM Banking System is to provide users with a secure and convenient banking experience.

To achieve this, the system offers an ATM interface that can be accessed through either a graphical user interface (GUI) or a command-line interface (CLI). This interface serves as a platform for users to easily carry out their transactions. We have implemented security measures to protect sensitive user information and ensure data privacy.

Our focus in developing this ATM Banking System is to offer users essential banking services and simplify their transaction processes. We prioritize user convenience, security, and data privacy to enhance their overall banking experience.

In summary, the development of this Java-based ATM Banking System aims to create a user-friendly and secure platform for users to perform banking tasks. The system includes features like managing accounts, transferring funds, checking balances, and processing transactions. Our goal is to provide users with a seamless and convenient banking experience while prioritizing their security and privacy.

**FEATURES**

The Java ATM banking system offers a range of important features to help users manage their accounts and conduct secure financial transactions. Let's explore these key functions:

* **Account Management**: Users can easily create new accounts and update their account details. They can also change their PINs for added security.
* **Fund Transfers**: The system allows users to transfer funds between their own accounts or to other accounts within the bank. This makes it simple to manage their finances and ensures that transfers are safe and efficient.
* **Balance Inquiries**: Users can check their account balances to keep track of their financial status. This provides them with real-time information on their available funds, helping them make informed financial decisions.
* **Transaction Processing**: The system handles withdrawals and deposits, ensuring that account balances are accurate and up-to-date. When a user requests a withdrawal, the specified amount is deducted from their balance. Similarly, when a deposit is made, the account balance is updated accordingly.
* **User Authentication**: The system employs strong user authentication measures, such as PIN verification, to ensure that only authorized users can access their accounts. This protects user information and prevents unauthorized access.

These features make the Java ATM banking system a user-friendly platform for efficient account management, secure fund transfers, balance inquiries, transaction processing, and user authentication. Users can confidently manage their finances and conduct banking operations with ease.

**CODE EXPLANATION**

* **BANK CLASS**

|  |
| --- |
| **import java.util.Random;**  **import java.util.HashMap;**  **import java.util.ArrayList;**  In the Bank class, the following imports were utilized: Random, which is responsible for generating random numbers used in creating a PIN; HashMap, used for storing a collection of usernames associated with their corresponding account objects; and ArrayList, used to hold a list of PINs. |
| **static protected HashMap<String, Account> accounts = new HashMap<>();**  The implementation of the Bank class incorporates a HashMap instead of an ArrayList to store accounts. This choice is made because the HashMap follows a key-value pair approach, allowing each account to be saved with a unique key. By using the username as the key, the program can easily access a specific account within the HashMap whenever necessary. In contrast, using an ArrayList would require iterating through the entire list to search for a specific account, which is less efficient. Therefore, the HashMap provides a more convenient and optimized solution for accessing and managing accounts in the Bank class. |
| **static protected ArrayList<StringBuilder> pins = new ArrayList<>();**  However, in the Bank class, an ArrayList is employed specifically for the storage of PINs. This choice is made due to the fact that the ArrayList is only required within the createPin(int pinLength) method. This method is responsible for generating a new PIN, and the use of the ArrayList within it is to validate whether the generated PIN already exists. This is accomplished by utilizing the "contains" method of the ArrayList, i.e., "pins.contains("pin generated")." If the generated PIN already exists within the list, it will not be created again, ensuring the uniqueness of each PIN. Therefore, the ArrayList serves its purpose effectively in this scenario by facilitating the validation of PIN uniqueness. |
| **protected static void createAccount(StringBuilder *pin*, String *name*) {**  **pins.add(pin);**  **accounts.put(name, new Account(name, pin));**  **}**  However, in the Bank class, an ArrayList is employed specifically for the storage of PINs. This choice is made due to the fact that the ArrayList is only required within the createPin(int pinLength) method. This method is responsible for generating a new PIN, and the use of the ArrayList within it is to validate whether the generated PIN already exists. This is accomplished by utilizing the "contains" method of the ArrayList, i.e., "pins.contains("pin generated")." If the generated PIN already exists within the list, it will not be created again, ensuring the uniqueness of each PIN. Therefore, the ArrayList serves its purpose effectively in this scenario by facilitating the validation of PIN uniqueness.  In the createAccount method of the Bank class, two parameters are accepted: the username and the pin. This method performs two actions. Firstly, it adds the provided pin to the pins list. Secondly, it utilizes both the username and pin to create an account object, which is then added to the accounts list. It is important to note that the username serves as the key for the corresponding account object within the list. By associating the username with the account object in this manner, the Bank class enables efficient retrieval and manipulation of specific accounts based on their unique usernames. |
| **static StringBuilder createPin(int *pinLength*) {**  **Random random = new Random();**  **StringBuilder pin = new StringBuilder();**  **boolean pinIsUnique = false;**  **while (!pinIsUnique) {**  **for (int i = 0; i < pinLength; i++) {**  **pin.append(random.nextInt(10));**  **}**  **if (pins.contains(pin)) {**  **pin.delete(0, pin.length());**  **} else {**  **pinIsUnique = true;**  **}**  **}**  **return pin;**  **}**  The method provided accepts an integer input that determines the desired length of the PIN. To generate the PIN, a StringBuilder object named "pin" is utilized. The choice of StringBuilder is based on the fact that String objects in Java are immutable, meaning that every time you modify a String, a new String object is created in memory. This process can be inefficient when constructing a new object for each string manipulation. In contrast, StringBuilder objects are mutable, allowing for modifications without the need to create new objects in memory.  Within the method, a boolean variable named "pinIsUnique" is initially set to false. This ensures that the application continues running until a unique PIN is generated. The PIN creation process involves concatenating randomly generated integers ranging from 0 to 9 using the random object. Once the PIN is constructed, it is checked to determine if it already exists within the "pins" collection. If a duplicate PIN is found, the pin creation process is restarted. Only when a unique PIN is generated, it is returned by the method. |

* **ACCOUNT CLASS**

|  |
| --- |
| **private int balance;**  **private String name;**  **private StringBuilder pin;**  The Account class includes several instances: balance, name, and pin.  The "balance" instance is utilized in the deposit, withdraw, and get balance methods of the Account class. It holds the current balance of the account and is involved in various financial operations.  The "name" instance is present within the Account class, although it is not currently utilized. If the account name becomes necessary in the future, it is available and can be accessed within this instance.  The "pin" instance represents the account's PIN and is of type StringBuilder. The choice of StringBuilder as the data type is because the PINs are created using a StringBuilder in the createPin method of the Bank class. The StringBuilder allows for efficient concatenation of the PIN digits during the PIN generation process. |
| **Account(String *name*, StringBuilder *tempPin*) {**  **this.name = name;**  **pin = tempPin;**  **}**  The following is a constructor for the Account class, specifically designed to be used in the createAccount method of the Bank class. In this constructor, the pin parameter is passed as the value for the account's pin, while the username parameter is used as the value for the account's username. This constructor ensures that when creating an account object within the Bank class, the appropriate pin and username values are assigned to the corresponding account attributes. |
| **void changeName(String *name*, String *newName*) {**  **accounts.put(newName, accounts.get(name));**  **accounts.remove(name);**  **this.name = newName;**  **}**  This method requires both the "name" and "new name" parameters. The previous name will be replaced, and the new name will be set as the current name for the account. The method pairs the new name with the previous name and creates the corresponding object. |
| **void changePin(StringBuilder *newPin*) {**  **pins.remove(pin);**  **pins.add(newPin);**  **this.pin = newPin;**  **}**  This method is responsible for updating the account's PIN. It accomplishes this by removing the old PIN from the list of pins and subsequently adding the new PIN to the list. |
| **void withdraw(int *withdrawAmount*) {**  **balance -= withdrawAmount;**  **}**  This method accepts the "withdrawAmount" argument and subtracts it from the account's balance. |
| **void deposit(int *cashAmount*) {**  **balance += cashAmount;**  **}**  This method is responsible for processing a deposit transaction. It takes the "cashAmount" argument and adds it to the account's balance. |
| **void transferFunds(String *receiver*, int *transferAmount*) {**  **accounts.get(receiver).balance += transferAmount;**  **balance -= transferAmount;**  **}**  This method enables the transfer of funds to another account. It requires the "receiver" parameter and the "transferAmount" parameter. When funds are transferred, the balance of the receiving account will be increased by the transferAmount, while the balance of the sending account will be decreased by the same amount. |
| **int getBalance() {**  **return balance;**  **}**  This method retrieves and returns the current balance of the account. |
| **String getPin() {**  **return pin.toString();**  **}**  This method returns the account's PIN by utilizing the `toString()` method. As the PIN is of the `StringBuilder` data type, invoking `toString()` on it will convert it into a `String`, allowing for its retrieval and subsequent return. |

* **ATMGUI CLASS**

|  |
| --- |
| **private JFrame frame = new JFrame();**  **private JLabel atm = new JLabel();**  **private static JTextField firstTextField, secondTextField ;**  **private JLabel firstRowLetters, secondRowLetters, thirdRowLetters;**  **private JPanel panel = new JPanel();**  **private JButton[] numberButtons = new JButton[10];**  **private JButton cancelButton = new JButton();**  **private JButton clearButton = new JButton();**  **private JButton enterButton = new JButton();**  **private String currentPage = "Start";**  **private final int BALANCE\_WITHDRAW\_LIMIT = 50000;**  **private String tempUsername;**  **private StringBuilder tempPin;**  **private  String username;**  **String receiver;**  **int transferAmount;**  The ATMGUI class requires the following imports and instances to construct interactive Graphical User Interfaces, as it serves as the location for the ATM's GUI.  The imports needed for GUI construction include JFrame, JLabel, JTextField, JButton, and JPanel. These instances are responsible for generating the necessary GUI components.  The currentPage property indicates the current page the user is on while using the program.  The balance limit is declared as final since it remains unchanged.  When creating a username and pin, the temporary variables tempUsername and tempPin are utilized. They serve as temporary storage locations for the user's input, as they may choose to continue or cancel the account creation process.  The username variable stores the account's username and is used to access the corresponding account object.  For fund transfers to another user, the receiver and transferAmount variables are utilized to specify the recipient's information and the amount being transferred, respectively. |
| **Below are the methods responsible for creating the panels of the ATM:**  start()  A screen shot of a computer  Description automatically generated with low confidence  signUpUsername()  A screenshot of a computer  Description automatically generated with low confidence  signUpPin()  A screen shot of a computer  Description automatically generated with low confidence  showSignUpDetails()  A screen shot of a machine  Description automatically generated with low confidence  enterUsername ()  A screen shot of a computer  Description automatically generated with low confidence  enterPin()  A screen shot of a machine  Description automatically generated with low confidence  showAtmFunctions()  A screen shot of a computer  Description automatically generated with low confidence  depositPage()  A screen shot of a machine  Description automatically generated with low confidence  withdrawPage()  A screenshot of a machine  Description automatically generated with low confidence  balancePage()  A screen shot of a machine  Description automatically generated with low confidence  transferFundsToPage()  A screen shot of a computer  Description automatically generated with low confidence  transferAmountPage()  A screen shot of a computer  Description automatically generated with low confidence  changeNamePage()  A screen shot of a machine  Description automatically generated with low confidence  newNamePage()  A screen shot of a computer  Description automatically generated with low confidence  changePinPage()  A screen shot of a machine  Description automatically generated with low confidence  newPinPage()  A screenshot of a computer  Description automatically generated with low confidence  showNewPin()  A screen shot of a machine  Description automatically generated with medium confidence |
| **void disableNumberButtons() {**  **for (int i = 0; i < 10; i++) {**  **numberButtons[i].setEnabled(false);**  **}**  **}**  This method deactivates the number buttons. |
| **void enableNumberButtons() {**  **for (int i = 0; i < 10; i++) {**  **numberButtons[i].setEnabled(true);**  **}**  **}**  This method activates the number buttons. |
| **@Override**  **public void actionPerformed(ActionEvent *e*)**  This is the section where we manage ActionEvents. |
| **if (e.getSource() == clearButton) {**  **firstTextField.setText("");**  **for (int i = 0; i < 10; i++) {**  **numberButtons[i].setEnabled(true);**  **}**  **enterButton.setEnabled(true);**  **}**  This is accompanied by the clearTextField method.  **void clearTextField() {**  **for (int i = 0; i < 10; i++) {**  **numberButtons[i].setEnabled(false);**  **}**  **enterButton.setEnabled(false);**  **}**  In the initial code, the method clears the text field and enables the number and enter buttons. This behavior is associated with another method, as it is used when information is displayed in the text field, such as when a user enters an invalid input. To proceed, the user must clear the text field because all other buttons except "cancel" are disabled. |
| **else if (e.getSource() == numberButtons[1]) {**  **firstTextField.setText(firstTextField.getText() + 1);**  **} else if (e.getSource() == numberButtons[2]) {**  **firstTextField.setText(firstTextField.getText() + 2);**  **} else if (e.getSource() == numberButtons[3]) {**  **firstTextField.setText(firstTextField.getText() + 3);**  **} else if (e.getSource() == numberButtons[4]) {**  **firstTextField.setText(firstTextField.getText() + 4);**  **} else if (e.getSource() == numberButtons[5]) {**  **firstTextField.setText(firstTextField.getText() + 5);**  **} else if (e.getSource() == numberButtons[6]) {**  **firstTextField.setText(firstTextField.getText() + 6);**  **} else if (e.getSource() == numberButtons[7]) {**  **firstTextField.setText(firstTextField.getText() + 7);**  **} else if (e.getSource() == numberButtons[8]) {**  **firstTextField.setText(firstTextField.getText() + 8);**  **} else if (e.getSource() == numberButtons[9]) {**  **firstTextField.setText(firstTextField.getText() + 9);**  **} else if (e.getSource() == numberButtons[0]) {**  **firstTextField.setText(firstTextField.getText() + 0);**  **}**  These if-else statements retrieve the source of the action and compare it to the number buttons. If the action originates from a particular number button, it retrieves the index of that button and inserts it into the text field. |
| **if (currentPage.equals("Start")) {**  **if (e.getSource() == numberButtons[1]) {**  **signUpUsername();**  **} else if (e.getSource() == numberButtons[2]) {**  **enterUsername();**  **}**  **}**  When the action is initiated from the number button 1, the program proceeds to the signUpUsername method, which displays the sign-up username page. Conversely, if the action originates from the number button 2 and the enterUsername() method is invoked, the login page will be shown. |
| **if (currentPage.equals("SignUpUsername")) {**  **if (e.getSource() == cancelButton) {**  **start();**  **}**  **if (e.getSource() == enterButton) {**  **if (firstTextField.getText().equals("") && secondTextField.getText().equals("")) {**  **firstTextField.setText("NO INPUT");**  **clearTextField();**  **} else if (firstTextField.getText().equals("")) {**  **if (accounts.containsKey(secondTextField.getText())) {**  **firstTextField.setText("ACCOUNT NAME ALREADY TAKEN");**  **clearTextField();**  **secondTextField.setText("");**  **} else {**  **tempUsername = secondTextField.getText();**  **signUpPin();**  **}**  **} else if (Integer.parseInt(firstTextField.getText()) > 26) {**  **firstTextField.setText("INVALID INPUT");**  **clearTextField();**  **} else {**  **secondTextField.setText(secondTextField.getText().concat**  **(String.valueOf((char) (Integer.parseInt(firstTextField.getText()) + 64))));**  **firstTextField.setText("");**  **}**  **}**  **}**  When the cancel button is pressed, it causes the program to revert back to the start page.  Upon pressing the enter button, the program performs several checks. Firstly, it verifies if there is no input, if the account username is already taken, or if the input is invalid. If any of these conditions are met, the corresponding information will be displayed in the first text field, which must be cleared by the user.  Alternatively, if there is no input but a username has already been selected, the program will temporarily store the username and proceed to the signUpPin method.  Lastly, if a valid input exists in the first text field, it will be converted to a letter displayed in the second text field. This conversion is achieved by adding 64 to the input number, as the ASCII values for letters A to Z range from 65 to 90. The resulting number is then converted to a char data type to represent the corresponding letter. Finally, the char value is converted to a string to enable concatenation and setting the resulting text into the text field. |
| **if (currentPage.equals("SignUpPin")) {**  **if (e.getSource() == cancelButton) {**  **signUpUsername();**  **}**  **if (e.getSource() == numberButtons[4]) {**  **tempPin = createPin(4);**  **showSignUpDetails(tempUsername, tempPin);**  **}**  **if (e.getSource() == numberButtons[6]) {**  **tempPin = createPin(6);**  **showSignUpDetails(tempUsername, tempPin);**  **}**  **}**  The cancel button in this context is intended to navigate back to the sign-up username page.  The user will be prompted to choose either 4 or 6 as the desired pin length. This selection will determine the length of the user's PIN and subsequently lead them to the sign-up details page. |
| **if (currentPage.equals("ShowSignUpDetails")) {**  **if (e.getSource() == cancelButton) {**  **signUpPin();**  **}**  **if (e.getSource() == enterButton) {**  **createAccount(tempPin, tempUsername);**  **start();**  **}**  **}**  In the sign-up details page, pressing the cancel button indicates the cancellation of the sign-up process, leading back to the sign-up pin page.  On the other hand, when the enter button is pressed in the sign-up details page, it signifies the confirmation of the entered details, prompting the program to create the account based on the provided information. |
| **if (currentPage.equals("EnterUsername")) {**  **if (e.getSource() == cancelButton) {**  **start();**  **}**  **if (e.getSource() == enterButton) {**  **if (firstTextField.getText().equals("") && secondTextField.getText().equals("")) {**  **firstTextField.setText("NO INPUT");**  **clearTextField();**  **}  else if (firstTextField.getText().equals("")) {**  **if (!accounts.containsKey(secondTextField.getText())) {**  **firstTextField.setText("ACCOUNT INVALID");**  **clearTextField();**  **secondTextField.setText("");**  **} else {**  **username = secondTextField.getText();**  **enterPin();**  **}**  **} else if (Integer.parseInt(firstTextField.getText()) > 26) {**  **firstTextField.setText("INVALID INPUT");**  **clearTextField();**  **} else {**  **secondTextField.setText(secondTextField.getText().concat**  **(String.valueOf((char) (Integer.parseInt(firstTextField.getText()) + 64))));**  **firstTextField.setText("");**  **}**  **}**  **}**  If the action originates from the cancel button, the program will navigate back to the start page.  When the enter button is pressed, the program verifies whether there is no input or if the input is invalid. If either condition is met, the corresponding information will be displayed in the first text field and must be cleared by the user. However, if there is no input but a username has already been selected, the program retrieves the username from the second text field and assigns it to the username instance. Subsequently, the program proceeds to the enterPin method.  In the case of a valid input in the first text field, the program converts the input to a letter displayed in the second text field. This conversion is achieved by adding 64 to the input number, as the ASCII values for letters A to Z range from 65 to 90. The resulting number is then converted to a char data type to represent the corresponding letter. Finally, the char value is converted to a string to enable concatenation when setting the text into the text field. |
| **else if (currentPage.equals("EnterPin")) {**  **if (e.getSource() == cancelButton) {**  **enterUsername();**  **}**  **if (e.getSource() == enterButton) {**  **if (firstTextField.getText().equals("")) {**  **firstTextField.setText("INPUT PIN");**  **clearTextField();**  **} else if (accounts.get(username).getPin().equals(firstTextField.getText())) {**  **showAtmFunctions();**  **} else {**  **firstTextField.setText("INVALID PIN");**  **clearTextField();**  **}**  **}**  **}**  If the cancel button is pressed during the pin entry process, the application will return to the enterUsername page.  When the enter button is pressed, error messages will be displayed if there is no input or if the entered pin is invalid. However, if the input pin is valid, it will be compared to the account pin by retrieving the username's pin and using the equals function. If the entered pin matches the account pin, the user will be logged in and directed to the showAtmFunctions page. |
| **if (currentPage.equals("ShowAtmFunctions")) {**  **if (e.getSource() == numberButtons[1]) {**  **depositPage();**  **} else if (e.getSource() == numberButtons[2]) {**  **withdrawPage();**  **} else if (e.getSource() == numberButtons[3]) {**  **balancePage();**  **} else if (e.getSource() == numberButtons[4]) {**  **transferFundsToPage();**  **} else if (e.getSource() == numberButtons[5]) {**  **changeNamePage();**  **} else if (e.getSource() == numberButtons[6]) {**  **changePinPage();**  **} else if (e.getSource() == numberButtons[7]) {**  **start();**  **}**  **}**  The page displayed will vary depending on the user's choice, which can be made by pressing the options corresponding to the number buttons. |
| **if (currentPage.equals("Deposit")) {**  **if (e.getSource() == cancelButton) {**  **showAtmFunctions();**  **}**  **if (e.getSource() == enterButton) {**  **if (firstTextField.getText().equals("")) {**  **firstTextField.setText("INPUT VALUE");**  **clearTextField();**  **} else if (Integer.parseInt(firstTextField.getText()) == 0) {**  **firstTextField.setText("ENTER VALID AMOUNT");**  **clearTextField();**  **} else {**  **accounts.get(username).deposit(Integer.parseInt(firstTextField.getText()));**  **showAtmFunctions();**  **}**  **}**  **}**  Pressing the cancel button will redirect users back to the showAtmFunctions screen.  On the other hand, when the enter button is pressed, the program will check if a valid amount is entered or if there is no input value. If a legitimate amount is provided, the deposit method associated with the account will be invoked to process the deposit. |
| **if (currentPage.equals("Withdraw")) {**  **if (e.getSource() == cancelButton) {**  **showAtmFunctions();**  **}**  **if (e.getSource() == enterButton) {**  **if (firstTextField.getText().equals("")) {**  **firstTextField.setText("INPUT VALUE");**  **clearTextField();**  **} else if (Integer.parseInt(firstTextField.getText()) > BALANCE\_WITHDRAW\_LIMIT) {**  **firstTextField.setText("LIMIT = 50000");**  **clearTextField();**  **} else if (Integer.parseInt(firstTextField.getText()) > accounts.get(username).getBalance()) {**  **firstTextField.setText("AMOUNT > BALANCE");**  **clearTextField();**  **} else if (Integer.parseInt(firstTextField.getText()) == 0) {**  **firstTextField.setText("ENTER VALID AMOUNT");**  **clearTextField();**  **} else {**  **accounts.get(username).withdraw(Integer.parseInt(firstTextField.getText()));**  **showAtmFunctions();**  **}**  **}**  **}**  By pressing the cancel button, users will be directed back to the showAtmFunctions screen.  On the other hand, when the enter button is pressed, the program evaluates several conditions. It checks for the presence of an input value, verifies if the amount exceeds the current balance, checks if the amount exceeds the withdrawal limit, and validates if the amount is within acceptable limits. If none of these conditions are met, the withdraw method associated with the account will be utilized to process the withdrawal. |
| **if (currentPage.equals("Balance")) {**  **if (e.getSource() == enterButton ) {**  **showAtmFunctions();**  **}**  **}**  When the enterButton is pressed on the balance page, it will proceed and return to the showAtmFunctions page. |
| **if (currentPage.equals("TransferFundsTo")) {**  **if (e.getSource() == cancelButton) {**  **showAtmFunctions();**  **}**  **if (e.getSource() == enterButton) {**  **if (firstTextField.getText().equals("") && secondTextField.getText().equals("")) {**  **firstTextField.setText("NO INPUT");**  **clearTextField();**  **}  else if (firstTextField.getText().equals("")) {**  **if (!accounts.containsKey(secondTextField.getText())) {**  **firstTextField.setText("ACCOUNT INVALID");**  **clearTextField();**  **secondTextField.setText("");**  **} else {**  **receiver = secondTextField.getText();**  **transferAmountPage();**  **}**  **} else if (Integer.parseInt(firstTextField.getText()) > 26) {**  **firstTextField.setText("INVALID INPUT");**  **clearTextField();**  **} else {**  **secondTextField.setText(secondTextField.getText().concat**  **(String.valueOf((char) (Integer.parseInt(firstTextField.getText()) + 64))));**  **firstTextField.setText("");**  **}**  **}**  **}**  If the action originates from the cancel button, the program will navigate back to the start page.  When the enter button is pressed, the program checks for the presence of input or if the input is invalid. If either condition is met, the corresponding information will be displayed in the first text field and must be cleared by the user. However, if there is no input but a username has already been selected, the program retrieves the username from the second text field and assigns it to the receiver instance. Subsequently, the program proceeds to the transferAmountPage method.  In the case of a valid input in the first text field, the program converts the input to a letter displayed in the second text field. This conversion is achieved by adding 64 to the input number, as the ASCII values for letters A to Z range from 65 to 90. The resulting number is then converted to a char data type to represent the corresponding letter. Finally, the char value is converted to a string to enable concatenation when setting the text into the text field. |
| **else if (currentPage.equals("TransferAmount")) {**  **if (e.getSource() == cancelButton) {**  **transferFundsToPage();**  **}**  **if (e.getSource() == enterButton) {**  **if (firstTextField.getText().equals("")) {**  **firstTextField.setText("NO INPUT");**  **clearTextField();**  **} else if (Integer.parseInt(firstTextField.getText()) > accounts.get(username).getBalance()) {**  **firstTextField.setText("AMOUNT > BALANCE");**  **clearTextField();**  **} else if (Integer.parseInt(firstTextField.getText()) == 0) {**  **firstTextField.setText("ENTER VALID AMOUNT");**  **clearTextField();**  **} else {**  **transferAmount = Integer.parseInt(firstTextField.getText());**  **accounts.get(username).transferFunds(receiver, transferAmount);**  **showAtmFunctions();**  **}**  **}**  **}**  Pressing the cancel button will navigate users back to the showAtmFunctions screen.  On the other hand, when the enter button is pressed, the program will check for several conditions. It will verify if there is no input, if the entered amount is not valid, or if the amount exceeds the available balance. If any of these conditions are met, appropriate error messages will be displayed. However, if none of these conditions are triggered, the program will retrieve the transfer amount from the text field and utilize the transferFunds method associated with the account to complete the fund transfer. |
| **else if (currentPage.equals("ChangeName")) {**  **if (e.getSource() == cancelButton) {**  **showAtmFunctions();**  **}**  **if (e.getSource() == enterButton) {**  **if (firstTextField.getText().equals("")) {**  **firstTextField.setText("INPUT PIN");**  **clearTextField();**  **} else if (accounts.get(username).getPin().equals(firstTextField.getText())) {**  **newNamePage();**  **} else {**  **firstTextField.setText("INVALID PIN");**  **clearTextField();**  **}**  **}**  **}**  If the cancel button is pressed during the pin entry process, the application will return to the enterUsername page.  When the enter button is pressed, error messages will be displayed if there is no input or if the entered pin is invalid. However, if the entered pin is valid, it will be compared to the account pin by retrieving the pin associated with the username and using the equals function. If the entered pin matches the account pin, the program will proceed to the newNamePage. |
| **else if (currentPage.equals("NewName")) {**  **if (e.getSource() == cancelButton) {**  **showAtmFunctions();**  **}**  **if (e.getSource() == enterButton) {**  **if (firstTextField.getText().equals("") && secondTextField.getText().equals("")) {**  **firstTextField.setText("NO INPUT");**  **clearTextField();**  **}  else if (firstTextField.getText().equals("")) {**  **if (accounts.containsKey(secondTextField.getText())) {**  **firstTextField.setText("USERNAME ALREADY TAKEN");**  **clearTextField();**  **secondTextField.setText("");**  **} else {**  **String newUsername = secondTextField.getText();**  **accounts.get(username).changeName(username, newUsername);**  **username  = newUsername;**  **start();**  **}**  **} else if (Integer.parseInt(firstTextField.getText()) > 26) {**  **firstTextField.setText("INVALID INPUT");**  **clearTextField();**  **} else {**  **secondTextField.setText(secondTextField.getText().concat**  **(String.valueOf((char) (Integer.parseInt(firstTextField.getText()) + 64))));**  **firstTextField.setText("");**  **}**  **}**  **}**  If the action originates from the cancel button, the program will navigate back to the start page.  When the enter button is pressed, the program checks for the presence of input or if the input is invalid. If either condition is met, the corresponding information will be displayed in the first text field and must be cleared by the user. However, if there is no input but a username has already been selected, the program retrieves the new username from the second text field and assigns it to the newUsername instance. This will result in the name being changed, and the user will be signed out from the program.  In the case of a valid input in the first text field, the program converts the input to a letter displayed in the second text field. This conversion is achieved by adding 64 to the input number, as the ASCII values for letters A to Z range from 65 to 90. The resulting number is then converted to a char data type to represent the corresponding letter. Finally, the char value is converted to a string to enable concatenation when setting the text into the text field. |
| **if (currentPage.equals("ChangePin")) {**  **if (e.getSource() == cancelButton) {**  **showAtmFunctions();**  **}**  **if (e.getSource() == enterButton) {**  **if (firstTextField.getText().equals("")) {**  **firstTextField.setText("INPUT PIN");**  **clearTextField();**  **} else if (accounts.get(username).getPin().equals(firstTextField.getText())) {**  **newPinPage();**  **} else {**  **firstTextField.setText("INVALID PIN");**  **clearTextField();**  **}**  **}**  **}**  If the cancel button is pressed while entering the pin, the application will return to the showAtmFunctions page.  When the enter button is pressed, error messages will be displayed if there is no input or if the entered pin is invalid. However, if the entered pin is valid, it will be compared to the account pin by retrieving the pin associated with the username and using the equals function. If the entered pin matches the account pin, the program will proceed to the newPinPage. |
| **if (currentPage.equals("NewPin")) {**  **if (e.getSource() == cancelButton) {**  **showAtmFunctions();**  **}**  **if (e.getSource() == numberButtons[4]) {**  **tempPin = createPin(4);**  **showNewPin();**  **}**  **if (e.getSource() == numberButtons[6]) {**  **tempPin = createPin(6);**  **showNewPin();**  **}**  **}**  The cancel button in this context allows the user to return to the showAtmFunctions page.  The user will have the option to choose a pin length of either 4 or 6 digits. The selected pin length will then determine the subsequent flow of the program, leading the user to the shownewPin page. |
| **if (currentPage.equals("ShowNewPin")) {**  **if (e.getSource() == cancelButton) {**  **newPinPage();**  **}**  **if (e.getSource() == enterButton) {**  **accounts.get(username).changePin(tempPin);**  **start();**  **}**  **}**  The cancel button functionality allows users to return to the newPinPage and restart the process from the beginning.  On the other hand, if the user chooses to proceed by pressing the enter button, the changePin method associated with the account will be invoked. This method will facilitate the process of changing the pin, and once completed, the user will be signed out from the system. |

**CONCLUSION**

This ATM banking system, developed using Java, is designed to provide users with a straightforward and convenient way to perform various banking tasks. The system offers a range of functionalities that allow users to create new accounts, change usernames and PINs, make deposits, withdraw funds, check account balances, and transfer money between accounts.

The main goal of this system is to offer a user-friendly experience, ensuring that individuals of all skill levels can easily navigate and utilize its features. Whether you are new to programming or have some experience, this program provides a simple and intuitive interface that makes it easy to understand and use.

Using this program, users can gain hands-on experience in Java programming and learn about the essential concepts involved in developing an ATM banking system. It serves as an educational tool, demonstrating how Java can be utilized to create functional and secure interfaces for managing financial transactions.

Whether you're looking to explore Java programming or want to enhance your existing skills, this ATM banking system offers a practical and accessible way to learn and apply programming concepts while providing practical solutions for everyday banking needs.