ATM Application and Report

CI701 Programming and Data Structures

Introduction

The main java code is missing elements like a deposit and withdraw function. To improve it we need to get it setup and working correctly first, meaning we need to add a deposit, withdraw and check balance function. Next, the layout is all done by hand which adds lots of extra code taking the attention away from the rest of the code. This means that if someone else were to receive this code it could be somewhat confusing for them. To improve this, we suggested that scene builder be used instead, allowing for quick layout changes and easier to read code.

The layout of the buttons was also not optimally designed so we should incorporate the design of a real ATM where the buttons are on either side of the screen making it more familiar to users. The next step was to add more functionality to the ATM such as a method to save accounts after logout, a method to add a new account and methods to generate random numbers for the user accounts and passwords.

To add and store an account we opted for an ArrayList as it was the simplest option for us. We would then need a function that would be able add the account details, and to iterate through the ArrayList and retrieve the account number and password. We would need to create a few files:

Design and Development

Scene Builder

The ATMApplication, Controller, Bank, and BankAccount files were all that was needed. Most of the code was taken from the original ATM-codebase to save time however, some was modified to fit our needs. Since we have opted for scene builder, we did not need a media or view files for this application, instead we would need an fxml file linked to the scene builder layout. This was done with the help of a tutorial on YouTube [1].

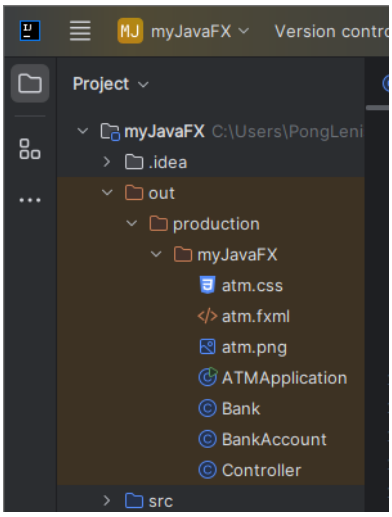

Figure 1: atm.fxml file

Once we had created an empty fxml file, we could right click it and open the file using scene builder. From there we could start dragging and dropping elements such as the buttons and text fields to create the applications GUI.

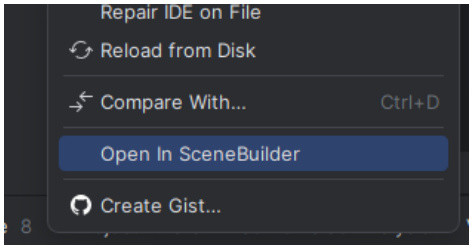

Figure 2: Using scene builder to open the fxml file

We are greeted with an empty scene builder template, but we will start by adding an anchor pane and within that anchor pane we will add all our interactive elements like the buttons and text fields. The final product is shown in Figure 3 below.

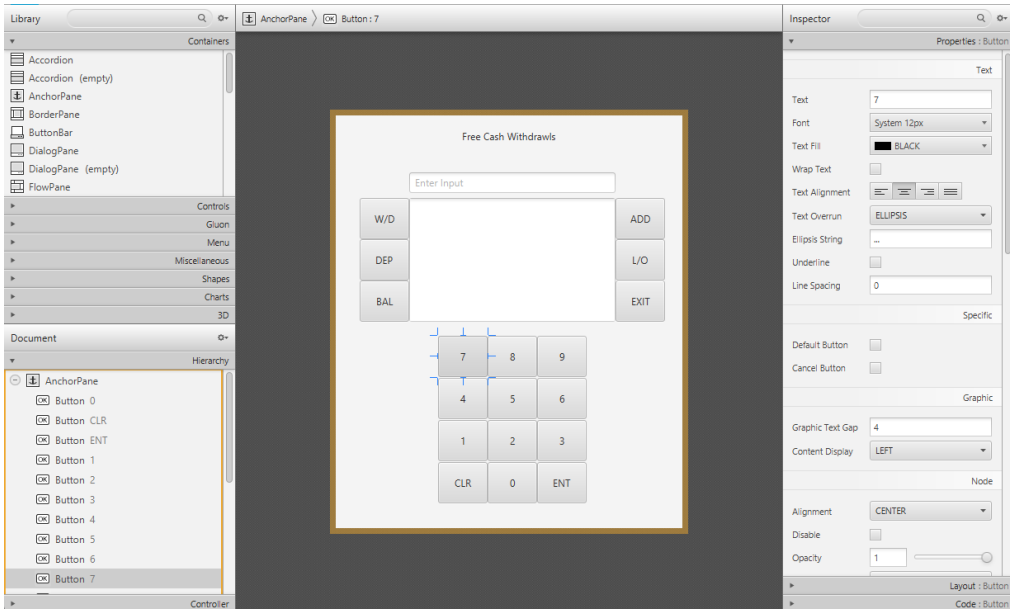

Figure 3: Scene builder GUI for ATM application

For this to work with our application we need to link it to our controller class file. We first create an empty controller class file in IntelliJ and then in the bottom left corner of scene builder we link this fxml file to that class file.

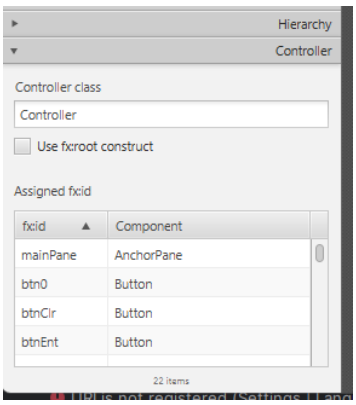

Figure 4: Linking the fxml file to the controller class file

The next step is to give all the buttons and text fields a unique ID so that we can control what each button does when pressed. Each element has a code section on the right-hand side, and we need to give it an ID and an action. This is shown in Figure 5 below.

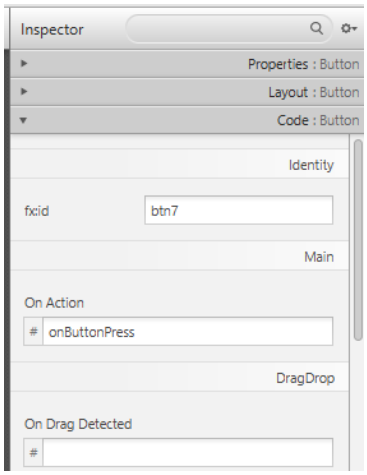

Figure 5: Giving an element an ID and action

When using event handlers with scene builder we need to add the following lines of code in Figure 6. This makes sure that all our buttons and text fields can be accessed by our application

```
public class Controller {  1 usage
    @FXML
    private Button btn0, btn1, btn2, btn3, btn4, btn5, btn6, btn7, btn8, btn9, btnClr, btnEnt;

    @FXML
    private Button withdrawBtn, depositBtn, balanceBtn, addAccountBtn, logoutBtn, exitBtn;

    @FXML
    private TextField userInput;

    @FXML
    private TextArea userResult;
```

Figure 6: Injecting fxml code using @FXML and fxmlLoader

```
<AnchorPane fx:id="mainPane" prefHeight="500.0" prefWidth="420.0" xmlns="http://javafx.com/javafx/23.0.1" xmlns:fx="http://javafx.co
   <children>
      <Button fx:id="btn0" alignment="CENTER" layoutX="184.0" layoutY="420.0" mnemonicParsing="false" onAction="#onButtonPress" prefH
      <Button fx:id="btnClr" alignment="CENTER" layoutX="124.0" layoutY="420.0" mnemonicParsing="false" onAction="#onButtonPress" pre
      <Button fx:id="btnEnt" alignment="CENTER" layoutX="244.0" layoutY="420.0" mnemonicParsing="false" onAction="#onButtonPress" pre
      <Button fx:id="btn1" alignment="CENTER" layoutX="124.0" layoutY="369.0" mnemonicParsing="false" onAction="#onButtonPress" prefH
      <Button fx:id="btn2" alignment="CENTER" layoutX="184.0" layoutY="369.0" mnemonicParsing="false" onAction="#onButtonPress" prefH
      <Button fx:id="btn3" alignment="CENTER" layoutX="244.0" layoutY="369.0" mnemonicParsing="false" onAction="#onButtonPress" prefH
      <Button fx:id="btn4" alignment="CENTER" layoutX="124.0" layoutY="318.0" mnemonicParsing="false" onAction="#onButtonPress" prefH
      <Button fx:id="btn5" alignment="CENTER" layoutX="184.0" layoutY="318.0" mnemonicParsing="false" onAction="#onButtonPress" prefH
      <Button fx:id="btn6" alignment="CENTER" layoutX="244.0" layoutY="318.0" mnemonicParsing="false" onAction="#onButtonPress" prefH
      <Button fx:id="btn7" alignment="CENTER" layoutX="124.0" layoutY="267.0" mnemonicParsing="false" onAction="#onButtonPress" prefH
      <Button fx:id="btn8" alignment="CENTER" layoutX="184.0" layoutY="267.0" mnemonicParsing="false" onAction="#onButtonPress" prefH
      <Button fx:id="btn9" alignment="CENTER" layoutX="244.0" layoutY="267.0" mnemonicParsing="false" onAction="#onButtonPress" prefH
      <TextField fx:id="userInput" layoutX="89.0" layoutY="69.0" prefHeight="25.0" prefWidth="250.0" promptText="Enter Input" />
      <TextArea fx:id="userResult" editable="false" layoutX="89.0" layoutY="100.0" prefHeight="150.0" prefWidth="250.0" wrapText="tru
      <Button fx:id="withdrawBtn" alignment="CENTER" layoutX="29.0" layoutY="100.0" mnemonicParsing="false" onAction="#onButtonPress"
      <Button fx:id="depositBtn" alignment="CENTER" layoutX="29.0" layoutY="150.0" mnemonicParsing="false" onAction="#onButtonPress"
      <Button fx:id="balanceBtn" alignment="CENTER" layoutX="29.0" layoutY="200.0" mnemonicParsing="false" onAction="#onButtonPress"
      <Button fx:id="exitBtn" alignment="CENTER" layoutX="339.0" layoutY="200.0" mnemonicParsing="false" onAction="#onButtonPress" pr
      <Button fx:id="addAccountBtn" alignment="CENTER" layoutX="339.0" layoutY="100.0" mnemonicParsing="false" onAction="#onButtonPre
      <Button fx:id="logoutBtn" alignment="CENTER" layoutX="339.0" layoutY="150.0" mnemonicParsing="false" onAction="#onButtonPress"
      <Label fx:id="label1" alignment="CENTER" contentDisplay="CENTER" prefHeight="50.0" prefWidth="420.0" text="Free Cash Withdrawls
   </children>
</AnchorPane>
```

Figure 7: FXML file with all the buttons and text fields

Controller Class

The “onButtonPress” is a method in the controller class file shown in Figure 8 below that contains a switch statement that handles what happens when each button is pressed.

```
public void onButtonPress(ActionEvent event) {  18 usages
    String ButtonId =((Button)event.getSource()).getId();

    switch(ButtonId){
        case "btn0":
            userInput.appendText("0");
            break;
```

Figure 8: onButtonPress method

We create an ‘event’ object that contains the details of the event that has occurred and the ‘event.getSource()’ retrieves the source of the event. In this case it would be the zero button. We type cast it to the ‘button’ type at the beginning allowing us to use button-specific attributes like ‘get.Id()’ to get the unique ID of the button [1]. For buttons that that the user uses to type, we just append the text in the user input text field seen in Figure 8. For buttons that handle depositing and withdrawing for example, we need to call a method when that button is pressed like in Figure 9.

```
case "btnEnt":
    handleEnter();
    break;
case "withdrawBtn":
    handleWithdraw();
    break;
case "depositBtn":
    handleDeposit();
    break;
```

Figure 9: Buttons to handle enter, withdraw and deposit

I kept the idea of having ‘states’ in the program to dictate what state the application is in such as, being logged in or entering account number. This was part of the original code base; however, I added my own to it to expand the programs functionality shown in Figure 10.

```
final String ACCOUNT_NUM = "account_num";  8 usages
final String PASSWORD = "password";  2 usages
final String LOGGED_IN = "logged_in";  6 usages
final String ADD_ACCOUNT = "adding_account";  3 usages
final String ACCOUNT_DISPLAY = "account_display";  2 usages

String state = ACCOUNT_NUM;  7 usages
```

Figure 10: States for the application

The last line in Figure 10 sets the default state when the application is first opened, this is part of a large switch statement that will be explained later in the report. It is then required to declare the variables shown in Figure 11 as they will be used throughout this class file by various methods such as depositing or withdrawing money. We make them private so that they are encapsulated and can only be accessed by the class in which they are declared [2].

```
private Bank bank;  7 usages
private BankAccount bankAcc;  20 usages
private String accountName;  6 usages
private String accountNumber;  8 usages
private double balance;  9 usages
```

Figure 11: Declaring private variables

We then need to include an 'initialize()' method that instantiates all the data in the fxml file like the buttons and the styling. This method is shown in Figure 12 below where it instantiates a new bank object, sets the state of the application to the specified state and calls a method to display a welcome message. In addition, it adds two tester bank accounts so that there will be already something stored in the array list to test the application with.

```
@FXML
public void initialize(){
    bank = new Bank();

    // Debugging purposes
    System.out.println("Initialising");
    System.out.println("Bank instance created");

    setState(ACCOUNT_NUM);
    displayMessage();

    // Tester accounts so that there is something to test with
    bank.addAccount("Gus De Sousa", "1234", "0000", 500);
    bank.addAccount("Alin Pacurar", "5678", "1111", 1000);
}

public void displayMessage(){  1 usage
    userResult.setText("Welcome, enter an account number " +
            "or exit by pressing the 'EXIT' button...");
}
```

Figure 12: Instantiating the application

Handling Enter

The 'handle enter' method is the longest method as it has several different things it can do depending on the state of the program. We mentioned the states earlier and this is where they come in useful.

```
public void handleEnter(){  1 usage
    String input = userInput.getText();
    switch(state){
        case ACCOUNT_NUM:
            accountNumber = input;
            userInput.clear();
            setState(PASSWORD);
            userResult.setText("Please enter your Password: ");
            break;
        case PASSWORD:
            String accountPassword = input;
            userInput.clear();
            if(bank.findAccount(accountNumber, accountPassword)){
                bankAcc = bank.setAccount(accountNumber, accountPassword);
                accountName = bankAcc.getAccountName();
                setState(LOGGED_IN);
                userResult.setText("Login Successful!\n" +
                        "Welcome "+accountName+"!\n"+
                        "Please enter an amount followed " +
                        "by an operator key or logout by " +
                        "pressing the 'L/O' button...");
            } else {
                userInput.clear();
                userResult.setText("Account or Password do not match. " +
                        "Please enter account number again...");
                setState(ACCOUNT_NUM);
            }
```

Figure 13: Handle enter method with switch statements

The states act as conditions for the switch statement in the handle enter method shown in figure 13 above. The first and default state of the program is called ACCOUNT_NUM, this will prompt the user to log in if they already have an account. If the user enters a number and presses enter, it will take that number and assign it to the accountNumber variable, clear the user input field and then set the state to PASSWORD.

This works in a similar way; it takes the user input and assigns it as a string to the accountPassword variable and clears the input field. There is an if statement this time as we need to make sure that there is an existing account with these credentials in the ArrayList. We call the 'findAccount()' method and pass in the current account number and password as arguments and if it finds a matching account then it will call the 'setAccount()' method to use that account during the logged in state. If the account number or password does not match an existing one, then it will give the user an error message and return to the ACCOUNT_NUM state.

It will also update the user on the screen giving them instructions on what they can do next in this logged in state such as, withdrawing or depositing money and logging out.

```
public void handleAddAccount(){  2 usages
    userResult.setText("Please type in your fullname " +
            "followed by the 'ENT' button...");
    setState(ADD_ACCOUNT);
}
```

Figure 14: handle add account method

The ADD_ACCOUNT state is triggered when the user presses the add button on the GUI and is displayed in figure 15 below. This runs a method as the first thing called 'handleAddAccount()' shown in figure 14 above.

What this method does is give the user a prompt to type in their name in the input field and press enter. It also sets the state to ADD_ACCOUNT.

```
case ADD_ACCOUNT:
    handleAddAccount();
    userInput.clear();
    if(input.matches("[a-zA-Z ]+")){
        bankAcc = new BankAccount("", "", "", 0);
        System.out.println("Bank account created");
        bankAcc.setAccountName(input);
        accountName = bankAcc.getAccountName();
        userInput.clear();

        int newAccNum = (int)(Math.floor(Math.random() * (9999 - 1000 + 1) + 1000));
        int newAccPass = (int)(Math.floor(Math.random() * (9999 - 1000 + 1) + 1000));
        userResult.setText("Hello "+accountName+"!\n"+
                "\n"+ "New account number: " +newAccNum+ "\n" +
                "New account password: " +newAccPass+ "\n" +
                "\n"+ "Please press 1 then enter to continue...");

        String accAsString = Integer.toString(newAccNum);
        String passAsString = Integer.toString(newAccPass);

        bankAcc.setAccountNumber(accAsString);
        bankAcc.setAccountPassword(passAsString);
        setState(ACCOUNT_DISPLAY);
    }
```

Figure 15: Adding account switch statement

When the user types in their name it will check if that string only contains letters using the line 'input.matches("[a-zA-Z]+")' in the if statement [*]. If that condition is true, then it will create a new empty bank account object and set the account name to the user's input. We then used a random number generator that will generate a random number between 1000 and 9999 [*], converts them to a string and then stores them in the new account object using the 'setAccount()' method.

The final line is to set the state to ACCOUNT_DISPLAY shown in figure 16 below. This state prompts the user to press 1 and hit enter to progress to the logged in stage of their new account. It'll display their account details and prompt them to deposit money into the account as by default, it is set to zero.

```
    break;
case ACCOUNT_DISPLAY:
    if(input.equals("1") && input.matches("[0-9]+")){
        userInput.clear();
        accountNumber = bankAcc.getAccountNumber();
        accountPassword = bankAcc.getAccountPassword();
        balance = bankAcc.getBalance();
        bank.addAccount(accountName, accountNumber, accountPassword, balance);
        userResult.setText("You're all set!\n" +
                "\n"+ "Account Name: "+accountName+"\n"+
                "Account #: "+accountNumber+"\n" +
                "Password: "+ accountPassword +"\n" +
                "Balance: $"+balance+"\n" +
                "\n"+ "- Enter amount followed by 'DEP' to deposit\n" +
                "- Select 'L/O' to logout");
        setState(LOGGED_IN);
    } else {
        userInput.clear();
        userResult.setText("Oops, something went wrong\n" +
                "Please try again!");
        setState(ADD_ACCOUNT);
    }
```

Figure 16: Account display state

Handling Withdraw

If the user is in a logged in state, then the handle withdraw method will be called when the user enters an amount and presses the withdraw button on the GUI.

```
public void handleWithdraw(){  1 usage
    double amount = Double.parseDouble(userInput.getText());
    if(state.equals(LOGGED_IN)){
        if(bankAcc == null){
            userResult.setText("Error: No account logged in");
        } else if(bankAcc.withdraw(amount)){
            balance = bankAcc.getBalance();
            userResult.setText("Withdrawn: $"+amount+
                    "Your remaining balance is: $"+balance+"\n");
            userInput.clear();
        } else {
            userResult.setText("You have insufficient funds!");
        }
    } else {
        userResult.setText("Error: Please log in or create an account first");
        userInput.clear();
        setState(ACCOUNT_NUM);
```

Figure 17: Handle withdraw method

It starts by parsing the input from a string to a double and then checks to see if the user is in a logged in state. If yes, then it will check if the bank account is not null and if so, prompt the user to log in. The else if statement calls the withdraw method from the BankAccount class and passes the amount entered by the user as the argument.

Handling Deposit

```
public void handleDeposit(){  1 usage
    double amount = Double.parseDouble(userInput.getText());
    if(state.equals(LOGGED_IN)){
        if(bankAcc == null){
            userResult.setText("Error: No account logged in!");
        }else if(bankAcc.deposit(amount)){
        balance = bankAcc.getBalance();

        for (int i = 0; i < bank.getAccounts().size(); i++) {
            if (bank.getAccounts().get(i).getAccountNumber().equals(bankAcc.getAccountNumber()) &&
                    bank.getAccounts().get(i).getAccountPassword().equals(bankAcc.getAccountPassword())) {
                bank.getAccounts().set(i, bankAcc);
            }
        }

        userResult.setText("Deposited: $"+amount+"\n" +
                "Your new balance is: $"+balance+"\n");
        userInput.clear();
        }
    } else {
        userResult.setText("Error: Please log in or create an account first");
        userInput.clear();
        setState(ACCOUNT_NUM);
    }
}
```

Figure 18: Handle deposit method

If the user is in the logged in state and has entered an amount, when the deposit key is pressed it will add the amount to the users account balance. Since the input type is a string, we need to parse that to a double data type [*]. Next, we have an if statement that checks whether the program is in a logged in state else, it will throw an error message that asks the user to log in or

make an account. If that condition is met, then there is another if statement that checks if the bank account is null, if it is null, it will prompt the user to log in.

The else if statement occurs when the above conditions are met, and it calls the deposit method from the BankAccount class passing the amount as the argument. The next line gets the updated balance of the bank account using the getAccount() method and stores it in the balance variable.

```
for (int i = 0; i < bank.getAccounts().size(); i++) {
    if (bank.getAccounts().get(i).getAccountNumber().equals(bankAcc.getAccountNumber()) &&
            bank.getAccounts().get(i).getAccountPassword().equals(bankAcc.getAccountPassword())) {
        bank.getAccounts().set(i, bankAcc);
    }
}
```

Figure 19: For loop to update account balance

The following block of code in figure 14 resolved an issue we were facing where after creating an account, the balance would be zero and the user was prompted to deposit money. The deposit would work, and the balance would update, however, if the user were to log out and log back in and check their balance, it would be zero. This was because we were updating the bank account object but not the account in the ArrayList with the new object.

What this block of code does is create a simple for loop that iterates through all the accounts thanks to a getter method and looks for the account that matches the credentials of the one currently logged in. The if statement is achieved when the bank account at index 'i' is equal to the currently logged in bank account based on the account number and password. If that if statement is satisfied, then it will overwrite and update the account details of the account at index 'i' using the set method [*]. It then tells the user that their balance has been updated and the shows them their new balance.

Bank Account Class

For this class, elements were largely kept the same with the addition of a few tweaks. We created a constructor class instead as this saves us having to write the same code multiple times. Moreover, for a bank account application with potentially multiple different accounts, this is a much more efficient way of carrying out this task [3].

```
private String accountName;  3 usages
private String accountNumber;  4 usages
private String accountPassword;  3 usages
private double balance;  6 usages

// Constructor
public BankAccount(String accountName, String accountNumber, String accountPassword, double balance){
    this.accountName = accountName;
    this.accountNumber = accountNumber;
    this.accountPassword = accountPassword;
    this.balance = balance;
}
```

Figure 20: Constructor method for bank account object

And because we have set the variables at the top of figure 13 to be private, we need to add getters and setters to access their information. By using getters and setters a controlled way is provided to read and modify the data whilst keeping it secured [4].

```java
// Getters and setters
public String getAccountName() { return accountName; }

public void setAccountName(String accountName) { this.accountName = accountName; }

public String getAccountNumber() { return accountNumber; }

public void setAccountNumber(String accountNumber) { this.accountNumber = accountNumber; }

public String getAccountPassword() { return accountPassword; }

public void setAccountPassword(String accountPassword) { this.accountPassword = accountPassword; }
```

Figure 21: Getter and Setters for account number and password

Figure 21 above shows the getters and setters needed for this application to work. To be able to login and create new accounts we need to be able to access and modify attributes such as the account number and password.

```java
public boolean withdraw(double amount){  1 usage
    if(amount > 0 && amount <= this.balance){
        this.balance -= amount;
        return true;
    }
    return false;
}

public boolean deposit(double amount){  1 usage
    if(amount > 0){
        this.balance += amount;
        return true;
    }
    return false;
}

public double getBalance(){  4 usages
    return this.balance;
}
```

Figure 22: Withdraw, deposit and getBalance methods

We make these methods in Figure 22 public so that they can be accessed by other classes in the application. We make the method return a Boolean value meaning it will either be true or false. We give the method one parameter which is the amount of money the user wants to withdraw.

The method first checks if the amount the user entered is above zero and if the amount entered is more than or equal to the amount of money in the account. Because we are using a logical operator, both cases must be true to proceed. If both cases are true, then we proceed to the amount being deducted from the accounts balance and the method will return true.

It is a similar process for the deposit method. We have an if statement that checks if the amount entered is above and only above zero. If that is true, the amount is added to the balance of the account and the method will return true.

The getBalance method is the simplest of them all. It returns a double value which, in this case, is the balance of the specified account.

Bank Class

The original code had an array that would only store a maximum of 10 accounts. For an application like this, that does not seem to be efficient. We went with an ArrayList instead as this has no limit on the number of accounts that can be added.

```
import java.util.ArrayList;

public class Bank {  2 usages
    ArrayList<BankAccount> accounts = new ArrayList<>();  6 usages

    public void addAccount(String accountName, String accountNumber, String accountPassword, double balance) {  3 usages
        BankAccount newAccount = new BankAccount(accountName, accountNumber, accountPassword, balance);
        accounts.add(newAccount);
        System.out.println("Account " + accountName + " successfully added! " + accountNumber + ", " + accountPassword + ", $" + balance);
    }
```

Figure 23: Add account ArrayList

To make an ArrayList we first need to import it into the class file. We then create an instance of the ArrayList and tell it what type of data we are storing in it and the name of that list [5]. We will store objects of the BankAccount type in this ArrayList.

We then need a method to add new accounts to the ArrayList called 'addAccount'. We also need parameters that will make up the account such as account number and password, name and balance. This is a public method meaning it can be accessed from outside this class file.

The first line in the method calls the constructor of the bank account class and populates the object with four arguments: name, number, password and balance. Once that has been done, it adds the account to the ArrayList with those details. The last line prints the account information to the console and is there for debugging purposes.

```
public boolean findAccount(String accountNumber, String accountPassword) {  1 usage
    for (BankAccount account : accounts) {
        if (account.getAccountNumber().equals(accountNumber) && account.getAccountPassword().equals(accountPassword)) {
            return true;
        }
    }
    return false;
}
```

Figure 24: A method to find an account with matching number and password

We then needed a method to find a specific account given the account number and password. It needed to return a Boolean value of true if it found an account with matching credentials. We use the account number and password as the parameters for this method.

A 'for-each' loop was used and the reason being is that it takes less steps and is more readable than a standard for loop [6][7]. In the 'for-each' loop bracket we have the datatype, the item we want to find and then the name of the ArrayList. In this case we are looking for an account, which has a BankAccount datatype in the accounts ArrayList.

While we are looping through the array, we are checking for a match using the if statement. We use the getter method to get the account number that the user has typed in and the equals attribute to compare the values and see if they match. We use the 'and' logical operator as we want both the account number and password to be a match for security. If an account is found it returns true, and the user is then logged in.

```
public BankAccount setAccount(String accountNumber, String accountPassword) {  1 usage
    for (BankAccount account : accounts) {
        if (account.getAccountNumber().equals(accountNumber) && account.getAccountPassword().equals(accountPassword)) {
            return account;
        }
    }
    return null;
}
```

Figure 25: Method to set the account when logging in

The code in Figure 25 works in the same way as Figure 24. It is a 'for-each' loop that is looking for an account with the BankAccount datatype in the accounts ArrayList. But when it finds the account, instead of returning a Boolean value, it will return the account details. This is used when the user has entered a matching account number and password and is logged in.

```
public ArrayList<BankAccount> getAccounts() {  4 usages
    return accounts;
}
```

Figure 26: Getter for the BankAccount ArrayList

To be able to retrieve the all the account details of the accounts in the ArrayList, we needed to set up a getter method shown in Figure 26 [8]. This allowed us to solve a problem that meant the balance was not updating after an account had been created and then logged out of.

References

[1] BroCode, "JavaFX GUI Course," *www.youtube.com*, Mar. 2021. https://www.youtube.com/watch?v=9XJicRt_FaI (accessed Nov. 2024).

[2] baeldung, "Java 'private' Access Modifier | Baeldung," *www.baeldung.com*, Sep. 03, 2019. https://www.baeldung.com/java-private-keyword (accessed Nov. 2024).

[3] BroCode, "Java Constructors," *www.youtube.com*, Sep. 2020. https://www.youtube.com/watch?v=lhf8gaUx4yU (accessed Nov. 2024).

[4] N. Ha Minh, "Best Practices for Java Getter and Setter - DZone," *dzone.com*, Oct. 2019. https://dzone.com/articles/java-getter-and-setter-basics-common-mistakes-and (accessed Nov. 2024).

[5] BroCode, "Java ArrayList," *www.youtube.com*, Oct. 2020. https://www.youtube.com/watch?v=1nRj4ALuw7A

[6] BroCode, "Java for-each Loop," *www.youtube.com*, Oct. 2020. https://www.youtube.com/watch?v=_IT8F5W0ZO4 (accessed Nov. 2024).

[7] B. Manos, "How to Search an Arraylist of Accounts for an Account with the Account Number Specified as a Parameter," *Stack Overflow*, Nov. 11, 2014. https://stackoverflow.com/questions/26864116/how-to-search-an-arraylist-of-accounts-for-an-account-with-the-account-number-sp (accessed Nov. 2024).

[8] D. Pryden, "Getters and Setters for ArrayLists in Java," *Stack Overflow*, Oct. 11, 2015. https://stackoverflow.com/questions/33060592/getters-and-setters-for-arraylists-in-java (accessed Nov. 2024).

[*] M. Aibin, “Convert String to Double in Java,” *Baeldung*, Aug. 03, 2019. https://www.baeldung.com/java-string-to-double (accessed Oct. 2024).

[*] G. Sachdeva, “Arraylist - Set/ Add Method Usage - Java,” *Stack Overflow*, Apr. 2017. https://stackoverflow.com/questions/9114564/arraylist-set-add-method-usage-java (accessed Nov. 2024).

[*] GeeksforGeeks, “Check If a String Contains Only Alphabets in Java,” *GeeksforGeeks*, Jul. 10, 2020. https://www.geeksforgeeks.org/check-if-a-string-contains-only-alphabets-in-java/ (accessed Nov. 2024).

[*] Educative, “How to Generate Random Numbers in Java,” *Educative: Interactive Courses for Software Developers*, Jul. 2022. https://www.educative.io/answers/how-to-generate-random-numbers-in-java