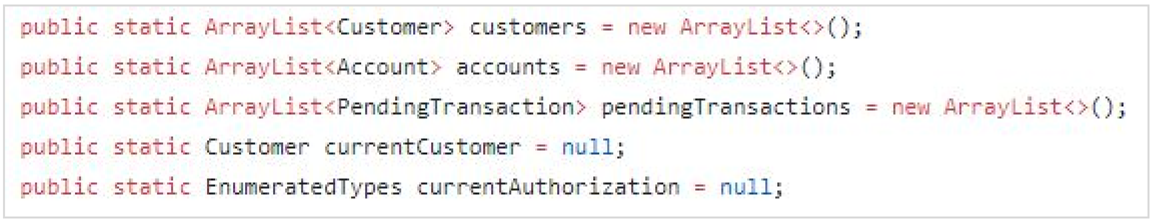
Table of Contents

1. Main.java

Within main the BankingDatabase.txt is read and written when the program opens and closes. Also included are the methods to manipulate the ArrayLists themselves as well as initialize the GUI.

1.1 Variables.



```
public static ArrayList<Customer> customers = new ArrayList<>();
public static ArrayList<Account> accounts = new ArrayList<>();
public static ArrayList<PendingTransaction> pendingTransactions = new ArrayList<>();
public static Customer currentCustomer = null;
public static EnumeratedTypes currentAuthorization = null;
```

1.2 ClosingThread() - Extends the Thread class. This class is runnable and responsible for writing the bank system data to the primary text file when the program closes. It gets the current data from the ArrayLists to write.

1.3 start() - Loads the FXML and runs the GUI.

```java
public void start(Stage primaryStage) throws Exception {
    Parent root = FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/LoginPage.fxml"));
    primaryStage.setTitle("Not Wells Fargo Banking System");
    primaryStage.setScene(new Scene(root, 1024, 768));
    primaryStage.show();
}
```

<u>1.4 main()</u> - Makes system startup calls.

```java
public static void main(String[] args) throws ParseException {
    Runtime.getRuntime().addShutdownHook(new ClosingThread());
    readDatabase();
    linkAccounts();
    launch(args);
}
```

<u>1.5 readDatabase()</u> - This method is used on startup to read in the data.txt for the banking system. The data is loaded into the appropriate ArrayList. Switch is used to split out the Customer, Checking, Savings, Loan, Transactions objects.

<u>1.6 removeAccount()</u> - Receives an Account to be removed from the accounts ArrayList.

```java
public static void removeAccount(Account accountToRemove){
    Customer customer = getCustomer(accountToRemove.getCustomerId());
    if (!customer.equals(null))
        customer.removeAccount(accountToRemove);
    accounts.remove(accountToRemove);
}
```

<u>1.7 linkAccounts()</u> - Links accounts to a customer.

```java
public static void linkAccounts(){
    for (Account account : accounts){
        for (Customer customer : customers){
            customer.addAccount(account);
        }
    }
}
```

<u>1.8 fetchTransactions()</u> - Recieves an accointId and returns the transactions for the account.

```java
public static ArrayList<PendingTransaction> fetchTransactions(int accountId){
    ArrayList<PendingTransaction> accountTransactions = new ArrayList();
    for (PendingTransaction pendingTransaction : pendingTransactions){
        if (accountId == pendingTransaction.getAccountID()){
            accountTransactions.add(pendingTransaction);
        }
    }
    return accountTransactions;
}
```

1.9 getCustomer() - Returns a customer from the customer ArrayList with the specified SSN.

1.10 addCustomer() - Receives a customer object and adds it to the customer ArrayList.

1.11 addAccount() - Recieves an account object and adds it to the accounts ArrayList. Calls linkAccounts to link the account tot the customer.

1.12 findCustomer() - Returns true/false if the customer exists in the system.

1.13 getDate() - Return the current date.

2. AccountsPackage

The Account Package includes all the code pertaining to customer, account and transaction data. Included are the following classes.

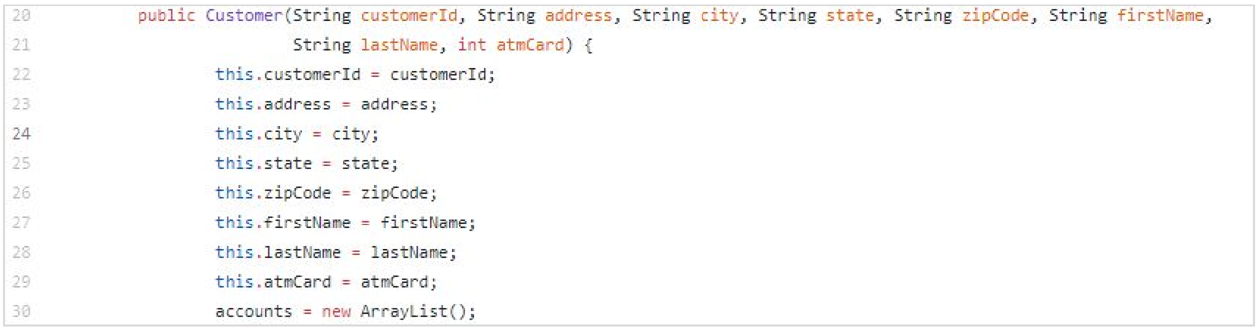
2.1 Customer.java

The customer class stores all the data needed for a customer in the system. All variables are protected and all methods are public. Every customers customerId is their social security number and thus unique.

2.1.1 Variables - The following variables are stored:

```java
        protected String customerId;
        protected String address;
        protected String city;
        protected String state;
        protected String zipCode;
        protected String firstName;
        protected String lastName;
        protected int atmCard = 0;
        protected ArrayList<Account> accounts;
```

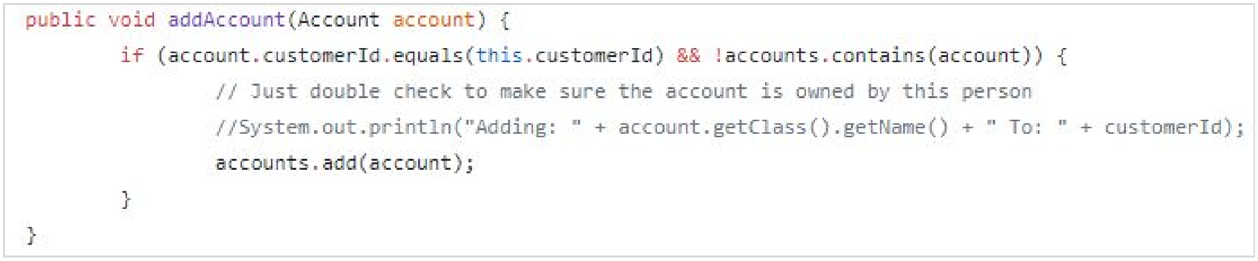
2.1.2 Constructor- The constructor will initialize all variables in the class.



```java
        public Customer(String customerId, String address, String city, String state, String zipCode, String firstName,
                        String lastName, int atmCard) {
                this.customerId = customerId;
                this.address = address;
                this.city = city;
                this.state = state;
                this.zipCode = zipCode;
                this.firstName = firstName;
                this.lastName = lastName;
                this.atmCard = atmCard;
                accounts = new ArrayList();
```

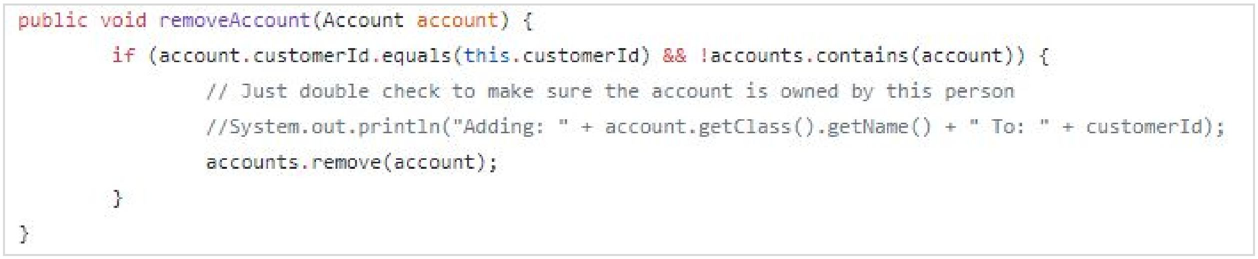
2.1.3 Getters and Setters - All variables have simple getter and setter methods which will not be listed here.

2.1.4 addAccount() - This method adds an Account item to the customers accounts ArrayList. A check is performed to ensure the account is owned by the customer.



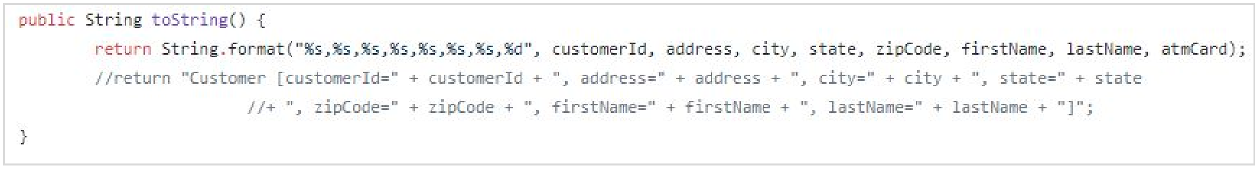
```java
public void addAccount(Account account) {
        if (account.customerId.equals(this.customerId) && !accounts.contains(account)) {
                // Just double check to make sure the account is owned by this person
                //System.out.println("Adding: " + account.getClass().getName() + " To: " + customerId);
                accounts.add(account);
        }
}
```

2.1.5 removeAccount() - This method removes an *Account* item to the customers accounts ArrayList. A check is performed to ensure the *Account* is owned by the *Customer*.



```java
public void removeAccount(Account account) {
        if (account.customerId.equals(this.customerId) && !accounts.contains(account)) {
                // Just double check to make sure the account is owned by this person
                //System.out.println("Adding: " + account.getClass().getName() + " To: " + customerId);
                accounts.remove(account);
        }
}
```

2.1.6 toString() - returns the *Customer* information in String format.

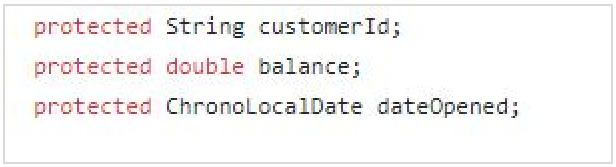


```java
public String toString() {
        return String.format("%s,%s,%s,%s,%s,%s,%s,%d", customerId, address, city, state, zipCode, firstName, lastName, atmCard);
        //return "Customer [customerId=" + customerId + ", address=" + address + ", city=" + city + ", state=" + state
                        //+ ", zipCode=" + zipCode + ", firstName=" + firstName + ", lastName=" + lastName + "]";
}
```

2.2 Account.java

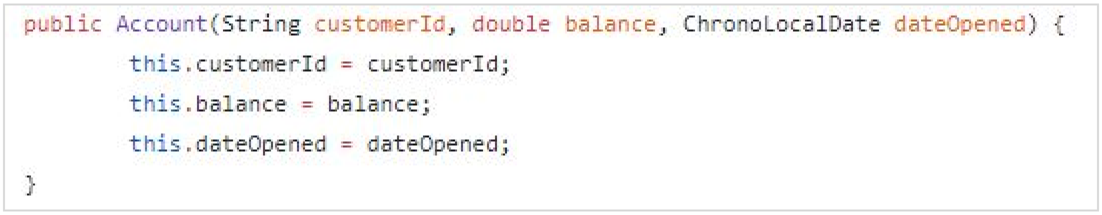
The Account class holds the account information. This class is abstract and extends to Loan, CheckingsAccount, SavingAccount classes. There cannot be an *Account* without a *Customer* who owns it.

2.2.1 Variables.



```
protected String customerId;
protected double balance;
protected ChronoLocalDate dateOpened;
```

2.2.2 Account() - constructor



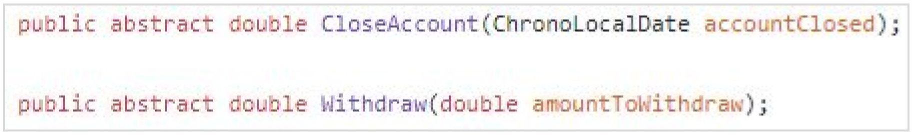
```
public Account(String customerId, double balance, ChronoLocalDate dateOpened) {
        this.customerId = customerId;
        this.balance = balance;
        this.dateOpened = dateOpened;
}
```

2.2.3 getCustomerId() - Returns *customerId*.

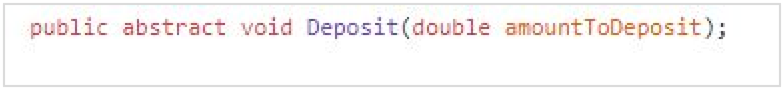
2.2.4 getBalance() - Returns *balance.*

.

2.2.5 Abstract Methods - The last 3 methods are abstract to be used by the extended classes.



```
public abstract double CloseAccount(ChronoLocalDate accountClosed);

public abstract double Withdraw(double amountToWithdraw);
```

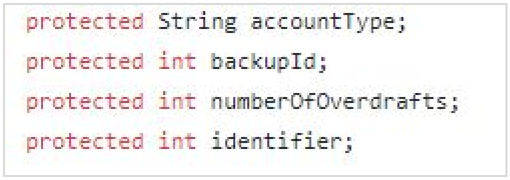


```
public abstract void Deposit(double amountToDeposit);
```

2.3 CheckingAccount.java

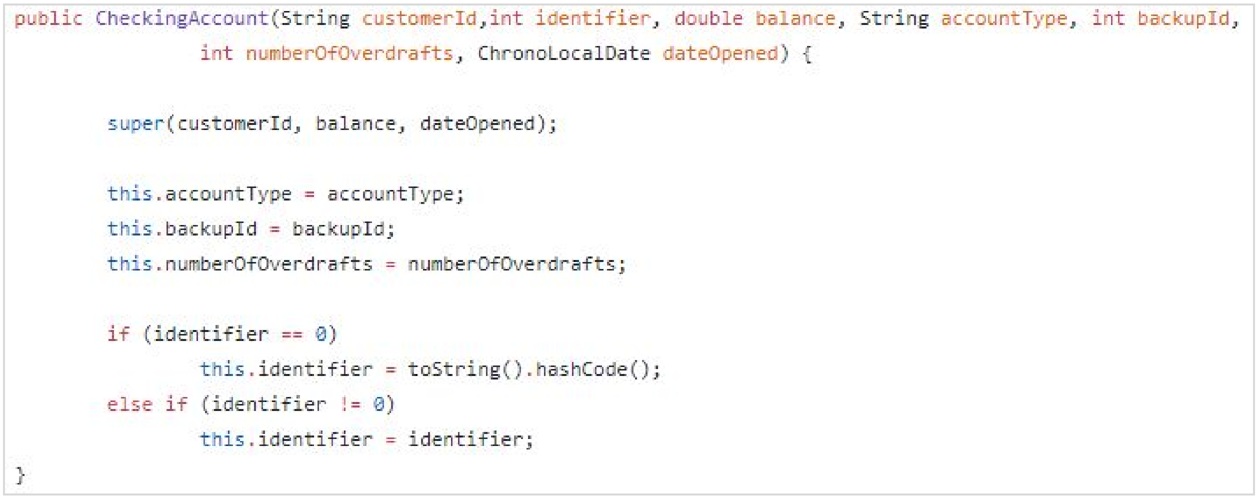
CheckingAccount is an extension of Account, and allows for customer to have a Checking Account.

2.3.1 Variables.

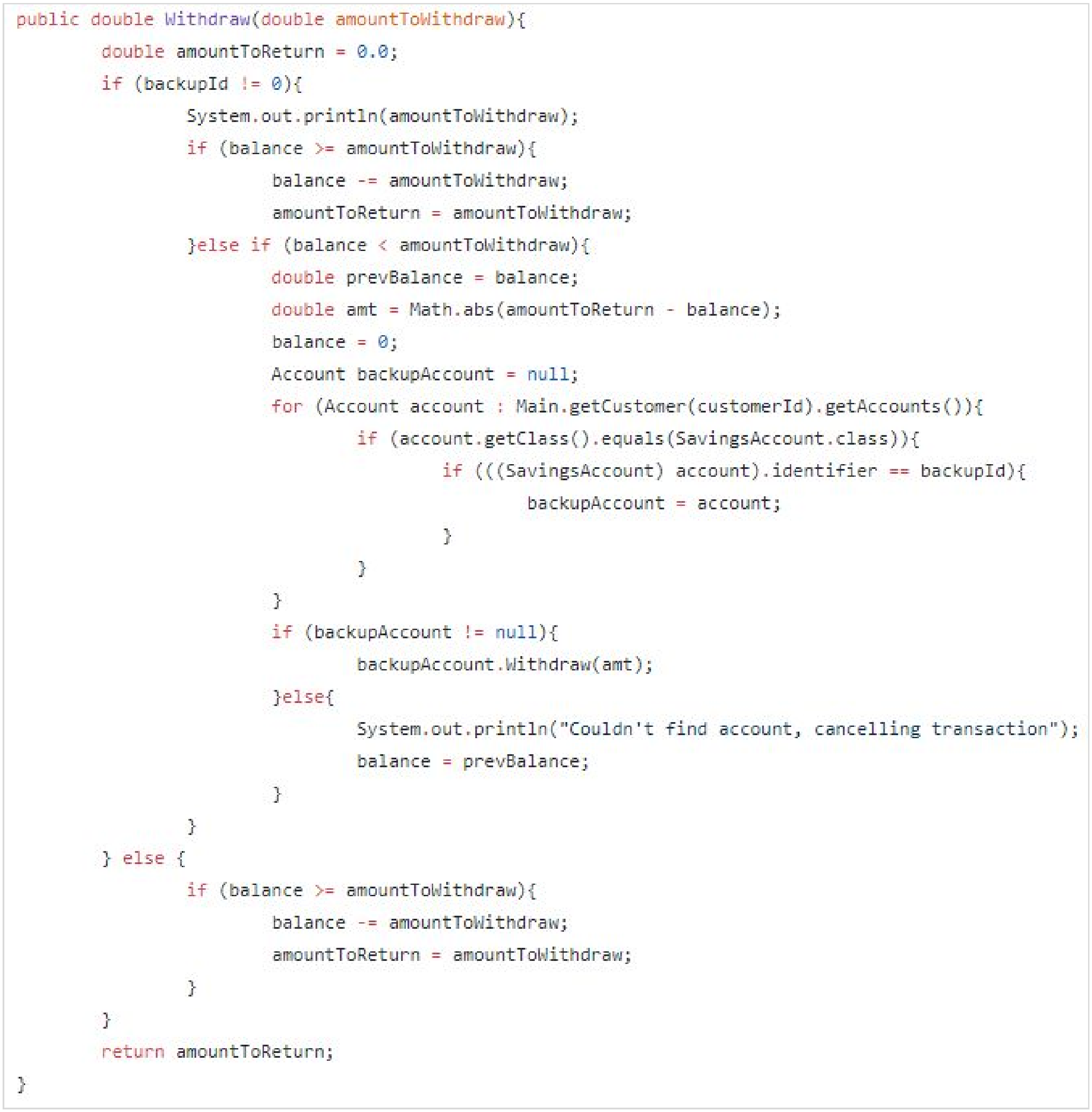


```
protected String accountType;
protected int backupId;
protected int numberOfOverdrafts;
protected int identifier;
```

2.3.2 Constructor - The constructor will initialize all variables in the class, and make sure the identifier is 0 or not.


```java
public CheckingAccount(String customerId,int identifier, double balance, String accountType, int backupId,
                int numberOfOverdrafts, ChronoLocalDate dateOpened) {

        super(customerId, balance, dateOpened);

        this.accountType = accountType;
        this.backupId = backupId;
        this.numberOfOverdrafts = numberOfOverdrafts;

        if (identifier == 0)
                this.identifier = toString().hashCode();
        else if (identifier != 0)
                this.identifier = identifier;
}
```

<u>2.3.3 Withdraw()</u> - This methods will get the customer withdraw, and make sure the account have enough balance to withdraw, and return the amount. If a backup Savings Account exists, check withdraw against balance. If Withdraw is greater than the balance it will withdraw overge from the backup account. Else check balance against withdraw amount and if over deny withdrawal. Returns amount withdrawn successfully.



```java
public double Withdraw(double amountToWithdraw){
        double amountToReturn = 0.0;
        if (backupId != 0){
                System.out.println(amountToWithdraw);
                if (balance >= amountToWithdraw){
                        balance -= amountToWithdraw;
                        amountToReturn = amountToWithdraw;
                }else if (balance < amountToWithdraw){
                        double prevBalance = balance;
                        double amt = Math.abs(amountToReturn - balance);
                        balance = 0;
                        Account backupAccount = null;
                        for (Account account : Main.getCustomer(customerId).getAccounts()){
                                if (account.getClass().equals(SavingsAccount.class)){
                                        if (((SavingsAccount) account).identifier == backupId){
                                                backupAccount = account;
                                        }
                                }
                        }
                        if (backupAccount != null){
                                backupAccount.Withdraw(amt);
                        }else{
                                System.out.println("Couldn't find account, cancelling transaction");
                                balance = prevBalance;
                        }
                }
        } else {
                if (balance >= amountToWithdraw){
                        balance -= amountToWithdraw;
                        amountToReturn = amountToWithdraw;
                }
        }
        return amountToReturn;
}
```

2.3.4 Deposit() - Receives an amount and adds it to balance.

2.3.5 CloseAccount() - Removes this *Account* from the system. Receives current time then removes the balance from the *Account* to be paid out to the customer. Pending transactions are checked for outstanding transactions. This account is sent to Main.removeAccount() to be removed from database ArrayList. Former balance is returned.

```
public double CloseAccount(ChronoLocalDate date){
        double amt = balance;
        balance = 0;
 ArrayList<PendingTransaction> pendingTransactions = Main.fetchTransactions(getAccountId());
 for (PendingTransaction pendingTransaction : pendingTransactions){
     Main.pendingTransactions.remove(pendingTransaction);
 }
        Main.removeAccount(this);
        return amt;
}
```

2.3.6 getAccountType() - Returns *accountType*.

2.3.7 toString() - Returns the Account as String in the following format:

```
String.format("%s,%d,%2.2f,%s,%d,%d,%s",customerId,identifier,balance,accountType,backupId,numberOfOverdrafts,df.format(dateOpened));
```

2.3.8 getAccountId() - Returns *identifier*.

2.4 SavingsAccount.java

SavingsAccount is an extension of accounts and holds the saving account information for customers.

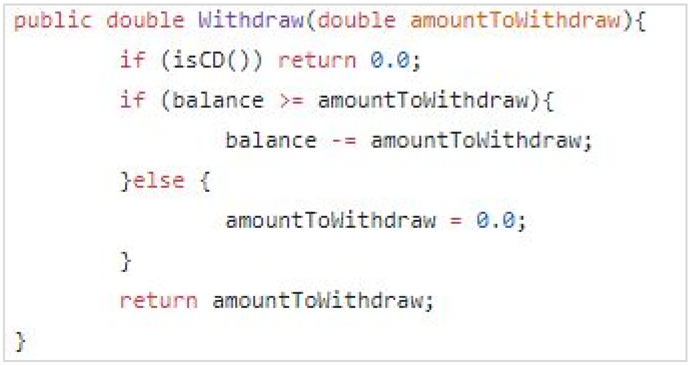
2.4.1 Variables.

```
protected static double currentInterestRate;
protected ChronoLocalDate CDDue;
protected int identifier;
```

2.4.2 Constructor.

```
public SavingsAccount(String customerId, int identifier, double balance, double currentInterestRate, ChronoLocalDate dateOpened, ChronoLocalDate CD
        super(customerId, balance, dateOpened);
        this.currentInterestRate = currentInterestRate;
        this.CDDue = CDDue;

        if (identifier == 0)
                this.identifier = toString().hashCode();
        else if (identifier != 0)
                this.identifier = identifier;
}
```

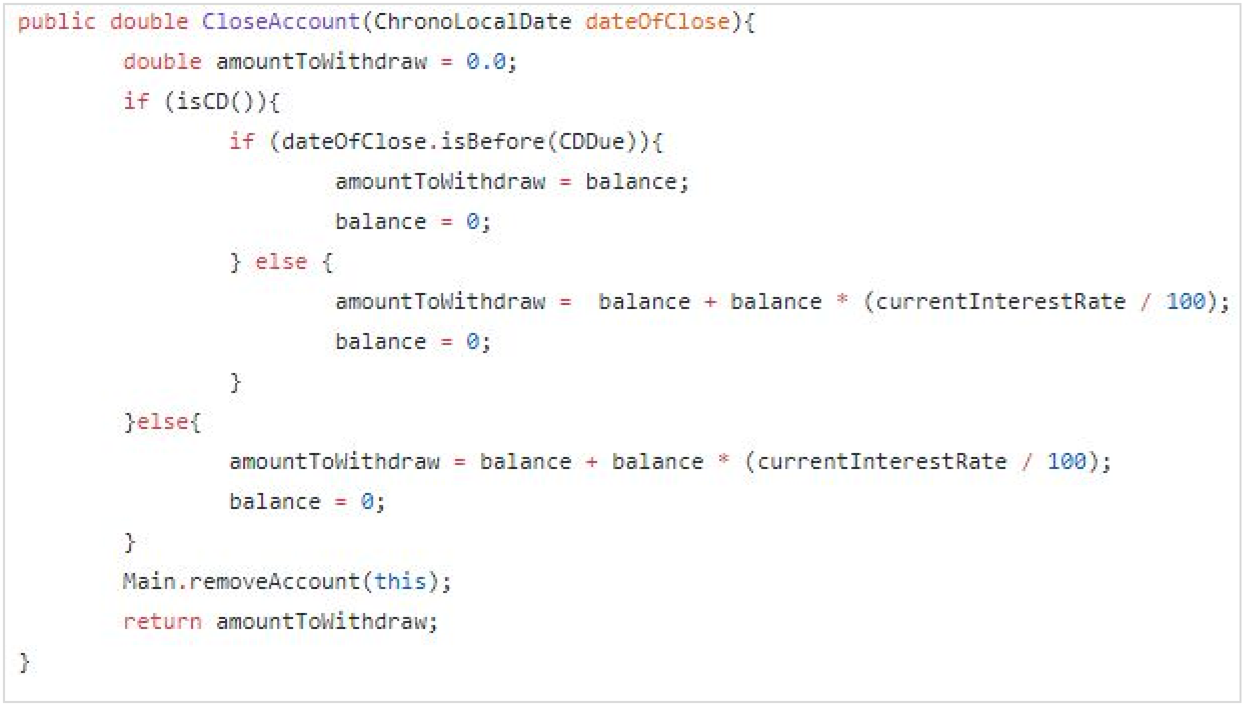
2.4.3 Withdraw() - If the account is a CD the methods returns 0. If the withdrawal is less than the balance the amount is withdrawn and balance updated, else return 0. Amount withdrawn is returned.



```
public double Withdraw(double amountToWithdraw){
        if (isCD()) return 0.0;
        if (balance >= amountToWithdraw){
                balance -= amountToWithdraw;
        }else {
                amountToWithdraw = 0.0;
        }
        return amountToWithdraw;
}
```

2.4.4 Deposit() - Receives amount to deposit and updates the balance.

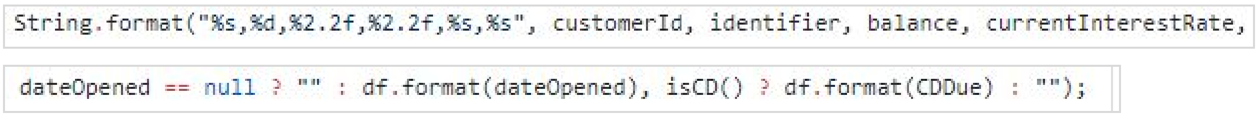
2.4.5 CloseAccount() - Closes this *Account.* If the Account is a CD the due date is checked. If closure call is before the CD due date no interest is returned. Else the close call is after CD due date the balance is returned with the calculated interest. Else the account is not a CD the balance is returned with occured interest. Balance is set to 0 and the account is sent to Main.removeAccount() to be removed from the database ArrayList.



```
public double CloseAccount(ChronoLocalDate dateOfClose){
        double amountToWithdraw = 0.0;
        if (isCD()){
                if (dateOfClose.isBefore(CDDue)){
                        amountToWithdraw = balance;
                        balance = 0;
                } else {
                        amountToWithdraw =  balance + balance * (currentInterestRate / 100);
                        balance = 0;
                }
        }else{
                amountToWithdraw = balance + balance * (currentInterestRate / 100);
                balance = 0;
        }
        Main.removeAccount(this);
        return amountToWithdraw;
}
```

2.4.6 isCD() - Returns boolean true if a due date exists as *CDDue.*

2.4.7 toString() - Returns this *SavingsAccount* information in String in the following format:



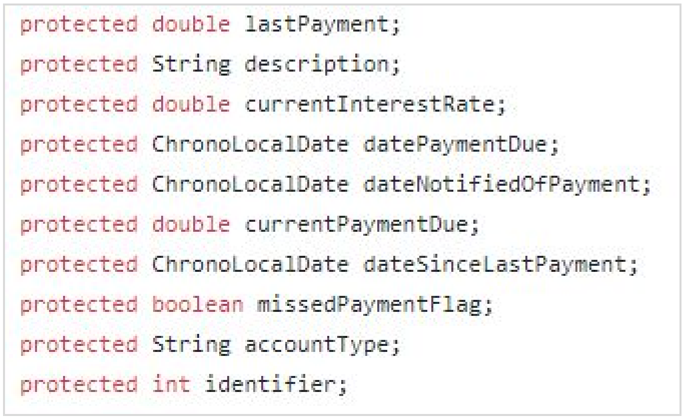
```
String.format("%s,%d,%2.2f,%2.2f,%s,%s", customerId, identifier, balance, currentInterestRate,
```

```
dateOpened == null ? "" : df.format(dateOpened), isCD() ? df.format(CDDue) : "");
```

2.5 Loan.java

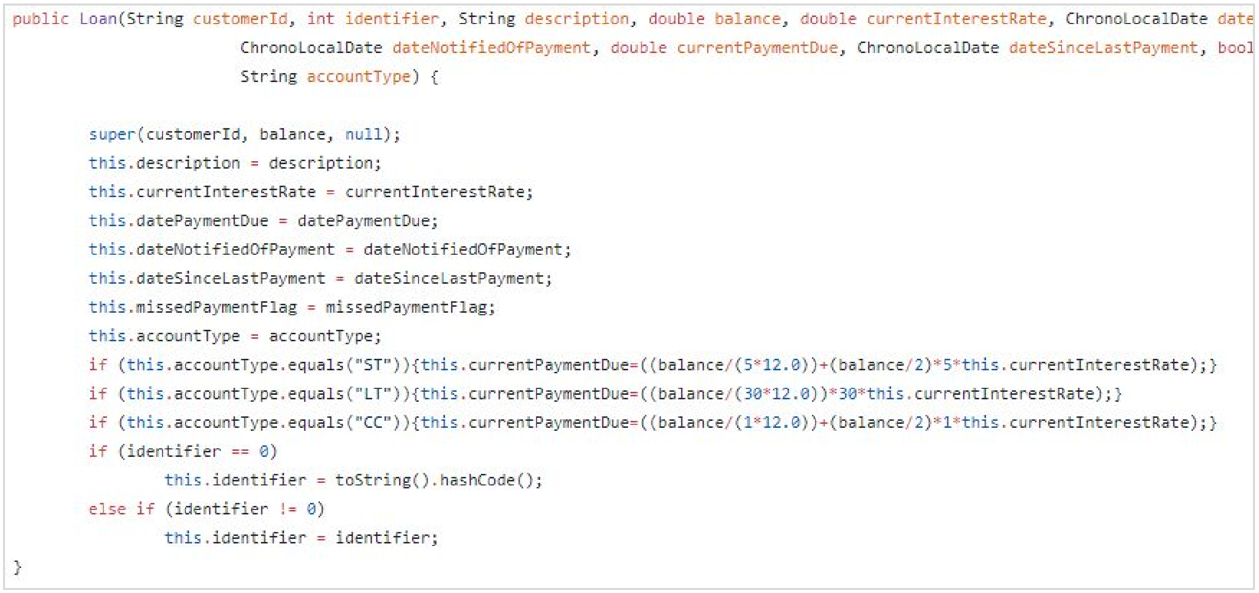
The loan class is an extension of accounts, that shows the loan information for customers. There are 3 loan types that each have different monthly payment calcualtions:
ST(Short Term) - *balance/(5*12.0))+(balance/2)*5*this.currentInterestRate*
LT(Long Term) - *balance/(30*12.0))*30*this.currentInterestRate*
CC(Credit Card) - *balance/(1*12.0))+(balance/2)*1*this.currentInterestRate*

<u>2.5.1 Variables.</u>



```
protected double lastPayment;
protected String description;
protected double currentInterestRate;
protected ChronoLocalDate datePaymentDue;
protected ChronoLocalDate dateNotifiedOfPayment;
protected double currentPaymentDue;
protected ChronoLocalDate dateSinceLastPayment;
protected boolean missedPaymentFlag;
protected String accountType;
protected int identifier;
```

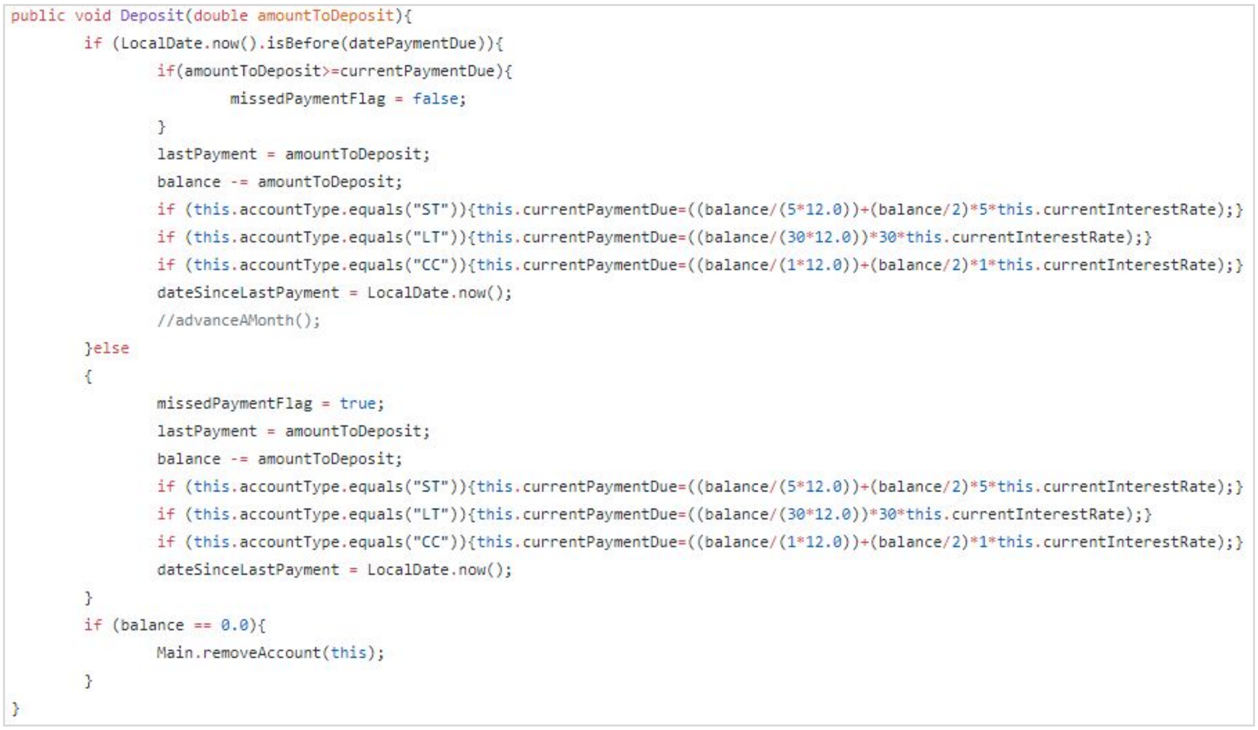
<u>2.5.2 Constructor</u> - Creates a loan object. The currentPaymentDue is set based on accountType. The payment calculations are performed within the constructor. The payment due is calculated based on accountType.



```
public Loan(String customerId, int identifier, String description, double balance, double currentInterestRate, ChronoLocalDate date
                    ChronoLocalDate dateNotifiedOfPayment, double currentPaymentDue, ChronoLocalDate dateSinceLastPayment, bool
                    String accountType) {

        super(customerId, balance, null);
        this.description = description;
        this.currentInterestRate = currentInterestRate;
        this.datePaymentDue = datePaymentDue;
        this.dateNotifiedOfPayment = dateNotifiedOfPayment;
        this.dateSinceLastPayment = dateSinceLastPayment;
        this.missedPaymentFlag = missedPaymentFlag;
        this.accountType = accountType;
        if (this.accountType.equals("ST")){this.currentPaymentDue=((balance/(5*12.0))+(balance/2)*5*this.currentInterestRate);}
        if (this.accountType.equals("LT")){this.currentPaymentDue=((balance/(30*12.0))*30*this.currentInterestRate);}
        if (this.accountType.equals("CC")){this.currentPaymentDue=((balance/(1*12.0))+(balance/2)*1*this.currentInterestRate);}
        if (identifier == 0)
                this.identifier = toString().hashCode();
        else if (identifier != 0)
                this.identifier = identifier;
}
```

<u>2.5.3 Deposit()</u> - Deposit is used to make payments on the Loan account. When the method is called it checks if the payment is late. If payment is not late it is calculated against the balance and payment due. A new payment due is calculated. If payment is late the late payment flag is

set to true. If balance is 0 the account will close itself.



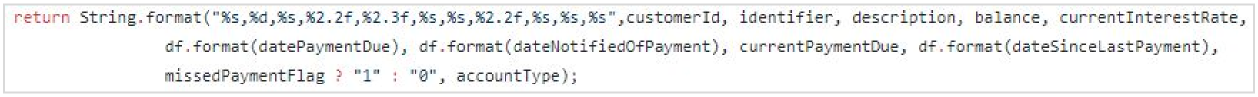
```
public void Deposit(double amountToDeposit){
        if (LocalDate.now().isBefore(datePaymentDue)){
                if(amountToDeposit>=currentPaymentDue){
                        missedPaymentFlag = false;
                }
                lastPayment = amountToDeposit;
                balance -= amountToDeposit;
                if (this.accountType.equals("ST")){this.currentPaymentDue=((balance/(5*12.0))+(balance/2)*5*this.currentInterestRate);}
                if (this.accountType.equals("LT")){this.currentPaymentDue=((balance/(30*12.0))*30*this.currentInterestRate);}
                if (this.accountType.equals("CC")){this.currentPaymentDue=((balance/(1*12.0))+(balance/2)*1*this.currentInterestRate);}
                dateSinceLastPayment = LocalDate.now();
                //advanceAMonth();
        }else
        {
                missedPaymentFlag = true;
                lastPayment = amountToDeposit;
                balance -= amountToDeposit;
                if (this.accountType.equals("ST")){this.currentPaymentDue=((balance/(5*12.0))+(balance/2)*5*this.currentInterestRate);}
                if (this.accountType.equals("LT")){this.currentPaymentDue=((balance/(30*12.0))*30*this.currentInterestRate);}
                if (this.accountType.equals("CC")){this.currentPaymentDue=((balance/(1*12.0))+(balance/2)*1*this.currentInterestRate);}
                dateSinceLastPayment = LocalDate.now();
        }
        if (balance == 0.0){
                Main.removeAccount(this);
        }
}
```

2.5.4 Getters and Setters - The following methods are simple getter and setter methods:
getAccountType()
getDatePaymentDue()
getCurrentInterestRate()
setCurrentInterestRate()
getCurrentPaymentDue()
getLastPayment()
getMissedPaymentFlag()
setMissedPaymentFlag()
getIdentifier()

2.5.5 advanceMonth() - This method is used to simulate 30 days passing. It will look at the current account balance, and interest rate and calculate a new balance. It will also give it a new due date.

2.5.6 updateInterestRate() - Function to update the interest rate. Takes in a new value for the new interest rate. 0.01 = 1%.

2.5.7 toString() - Returns the loan account data in the following format:

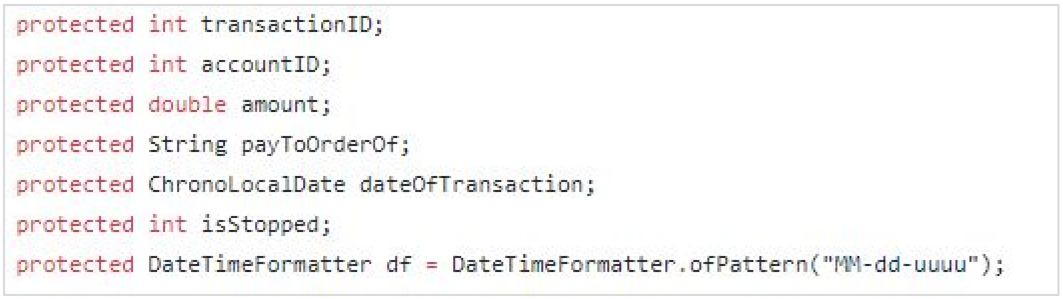


```
return String.format("%s,%d,%s,%2.2f,%2.3f,%s,%s,%2.2f,%s,%s,%s",customerId, identifier, description, balance, currentInterestRate,
                df.format(datePaymentDue), df.format(dateNotifiedOfPayment), currentPaymentDue, df.format(dateSinceLastPayment),
                missedPaymentFlag ? "1" : "0", accountType);
```

2.6 PendingTransaction.java

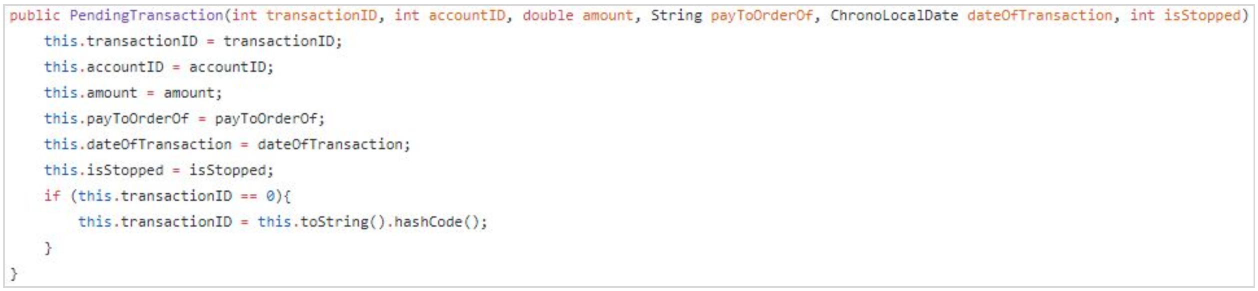
This class handles pending transactions against accounts. It include functionality to stop payments if they have not been honored yet.

2.6.1 Variables.



```java
protected int transactionID;
protected int accountID;
protected double amount;
protected String payToOrderOf;
protected ChronoLocalDate dateOfTransaction;
protected int isStopped;
protected DateTimeFormatter df = DateTimeFormatter.ofPattern("MM-dd-uuuu");
```

2.6.2 Constructor.

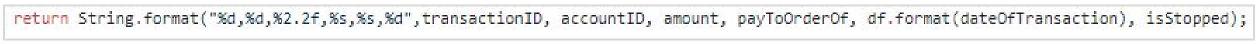


```java
public PendingTransaction(int transactionID, int accountID, double amount, String payToOrderOf, ChronoLocalDate dateOfTransaction, int isStopped)
    this.transactionID = transactionID;
    this.accountID = accountID;
    this.amount = amount;
    this.payToOrderOf = payToOrderOf;
    this.dateOfTransaction = dateOfTransaction;
    this.isStopped = isStopped;
    if (this.transactionID == 0){
        this.transactionID = this.toString().hashCode();
    }
}
```

2.6.3 Getters and setters - included simple getters and setters are as follows:

getAccountID()
getTransactionID()
getAmount()
getPayToOrderOf()
getDateOfTransaction()
getIsStopped()
setIsStopped()

2.6.4 toString() - Returns a String of the PendingTranaction data in the following format:



```java
return String.format("%d,%d,%2.2f,%s,%s,%d",transactionID, accountID, amount, payToOrderOf, df.format(dateOfTransaction), isStopped);
```

3. ControllerPackage

The Controller class will access the input boxes and other various items from the fxml pages. The pages should contain fx:id to be able to be accessed on this page using event listeners.

3.1 MainController.java

This controller handles the initial login of a Manager Teller or Customer

3.1.1 Variables:

```
public Button loginButton;


public TextField userLoginText;
public TextField userPassText;
```

3.1.2 function() - This used for changing the gui pages and setting the new scene.

```
private void function(Parent parent, ActionEvent event) {
    Scene homePageScene = new Scene(parent);
    Stage app_stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
    app_stage.setScene(homePageScene);
```

3.1.3 login(): loads the FXML and get user information and separate the login is Teller or manager or customer.

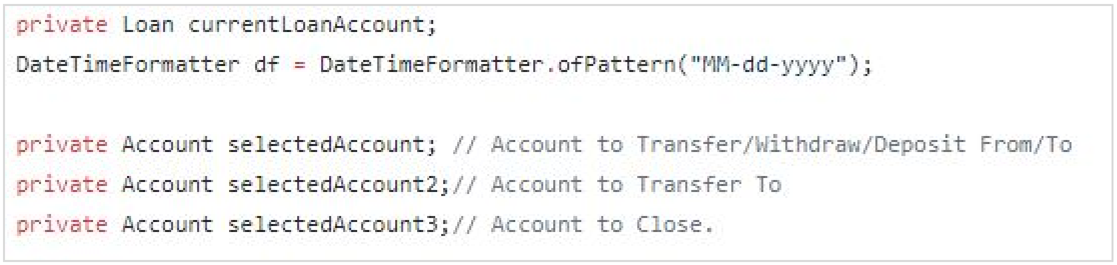
```
void login(ActionEvent event) throws IOException {
    if (userLoginText.getText().equals(null) || userPassText.getText().equals(null) ||
            userLoginText.getText().trim().equals("") || userPassText.getText().trim().equals("")) return;
    String user = userLoginText.getText().toLowerCase();
    String pass = userPassText.getText().toLowerCase();
    if (user.equals("teller") && pass.equals("1")) {
        Main.currentAuthorization = EnumeratedTypes.TELLER;
        function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/TellerMainMenu.fxml")), event);
    }else if (user.equals("manager") && pass.equals("1")) {
        Main.currentAuthorization = EnumeratedTypes.MANAGER;
        function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/ManagerMainMenu.fxml")), event);
    }else if (pass.equals("1") && !user.equals(null) && !user.equals("") && Main.findCustomer(user)) {
        Main.currentAuthorization = EnumeratedTypes.CUSTOMER;
        Customer customer = Main.getCustomer(user);
        Main.currentCustomer = customer;
        function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/Customer.fxml")), event);
    } else {
        userLoginText.setText("");
        userPassText.setText("");
    }
}
```

3.1.4 initialize(`URL arg0, ResourceBundle arg1`)

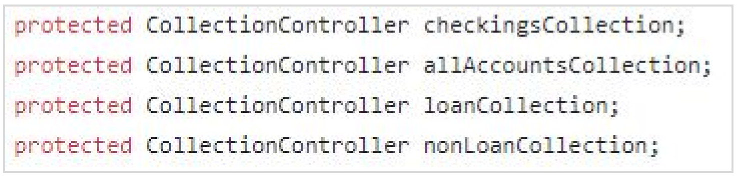
3.2 SubController.java

The SubController handles all the overall user functionality for the system.

3.2.1 Variables - In addition to the below image there are many Button, ComboBox, TextField, TextArea, ListView, Pane and Label variables as well.

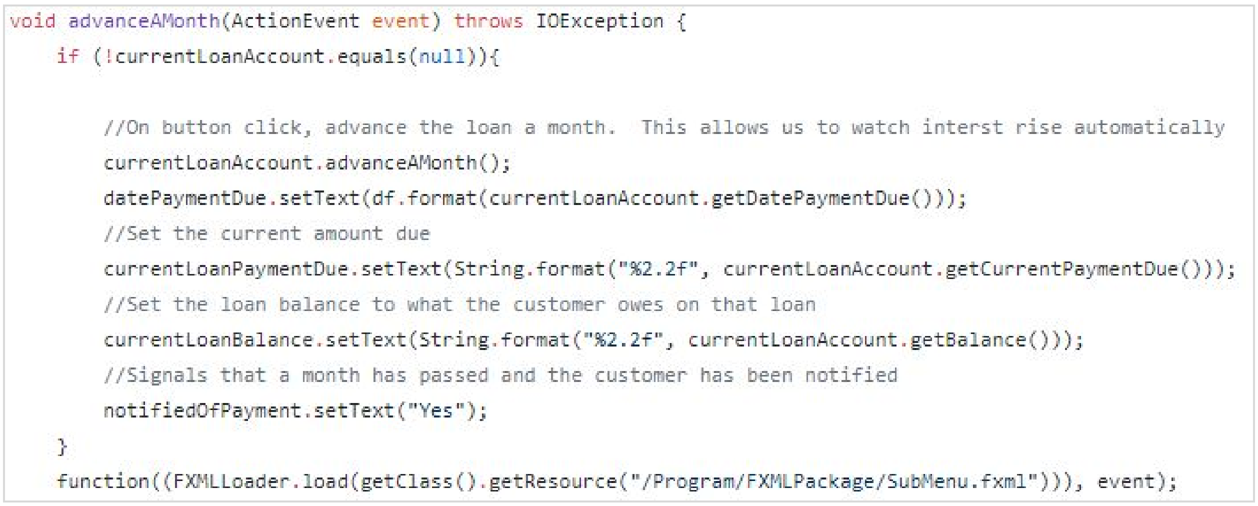


```java
private Loan currentLoanAccount;
DateTimeFormatter df = DateTimeFormatter.ofPattern("MM-dd-yyyy");

private Account selectedAccount; // Account to Transfer/Withdraw/Deposit From/To
private Account selectedAccount2;// Account to Transfer To
private Account selectedAccount3;// Account to Close.
```



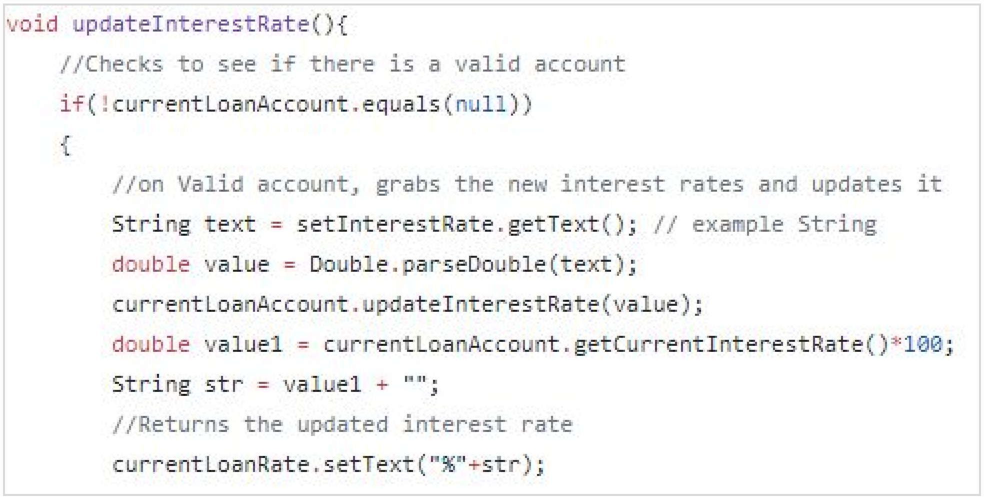
```java
protected CollectionController checkingsCollection;
protected CollectionController allAccountsCollection;
protected CollectionController loanCollection;
protected CollectionController nonLoanCollection;
```

3.2.2 advanceAMonth() - On button click, advance the loan a month. This allows us to watch interest rise automatically,Set the current amount due,Set the loan balance to what the customer owes on that loan,Signals that a month has passed and the customer has been notified.



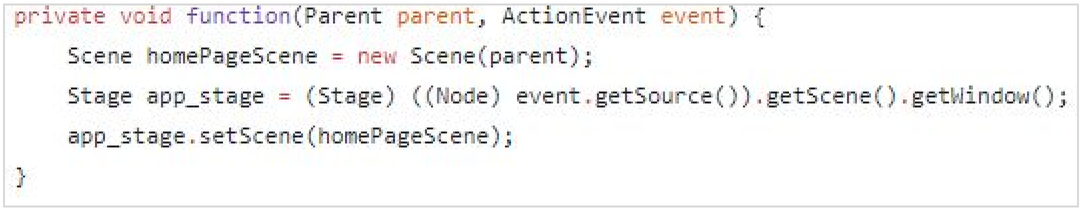
```java
void advanceAMonth(ActionEvent event) throws IOException {
    if (!currentLoanAccount.equals(null)){

        //On button click, advance the loan a month.  This allows us to watch interst rise automatically
        currentLoanAccount.advanceAMonth();
        datePaymentDue.setText(df.format(currentLoanAccount.getDatePaymentDue()));
        //Set the current amount due
        currentLoanPaymentDue.setText(String.format("%2.2f", currentLoanAccount.getCurrentPaymentDue()));
        //Set the loan balance to what the customer owes on that loan
        currentLoanBalance.setText(String.format("%2.2f", currentLoanAccount.getBalance()));
        //Signals that a month has passed and the customer has been notified
        notifiedOfPayment.setText("Yes");
    }
    function((FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/SubMenu.fxml"))), event);
```

3.2.3 updateInterestRate() - Check if the account is valid or not, on valid account, it grabs the new interest rates and updates it, then returns the rate.



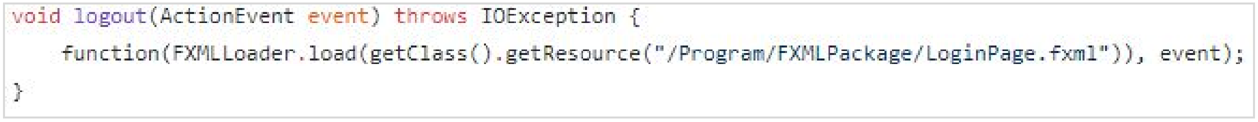
```
void updateInterestRate(){
    //Checks to see if there is a valid account
    if(!currentLoanAccount.equals(null))
    {
        //on Valid account, grabs the new interest rates and updates it
        String text = setInterestRate.getText(); // example String
        double value = Double.parseDouble(text);
        currentLoanAccount.updateInterestRate(value);
        double value1 = currentLoanAccount.getCurrentInterestRate()*100;
        String str = value1 + "";
        //Returns the updated interest rate
        currentLoanRate.setText("%"+str);
```

3.2.4 function() - This used for changing the gui pages and setting the new scene.



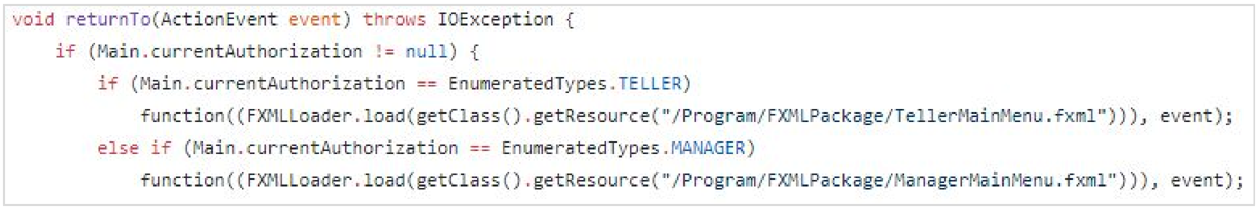
```
private void function(Parent parent, ActionEvent event) {
    Scene homePageScene = new Scene(parent);
    Stage app_stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
    app_stage.setScene(homePageScene);
}
```

3.2.5 logout() - This used for user log out the page.



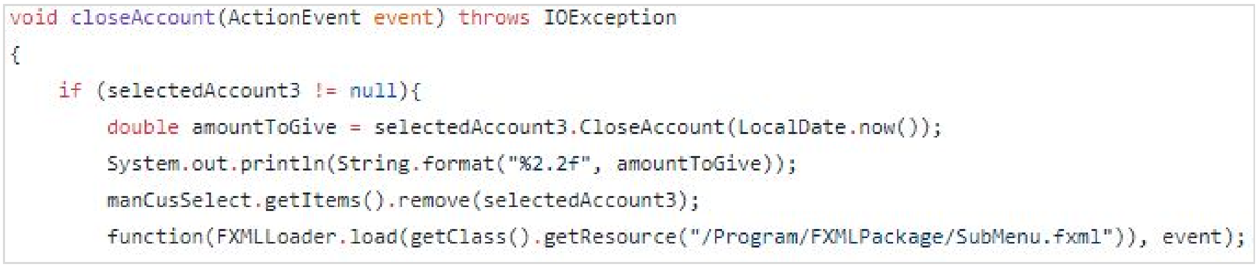
```
void logout(ActionEvent event) throws IOException {
    function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/LoginPage.fxml")), event);
}
```

3.2.6 returnTo() - This used for Gui return to manager page or teller page.



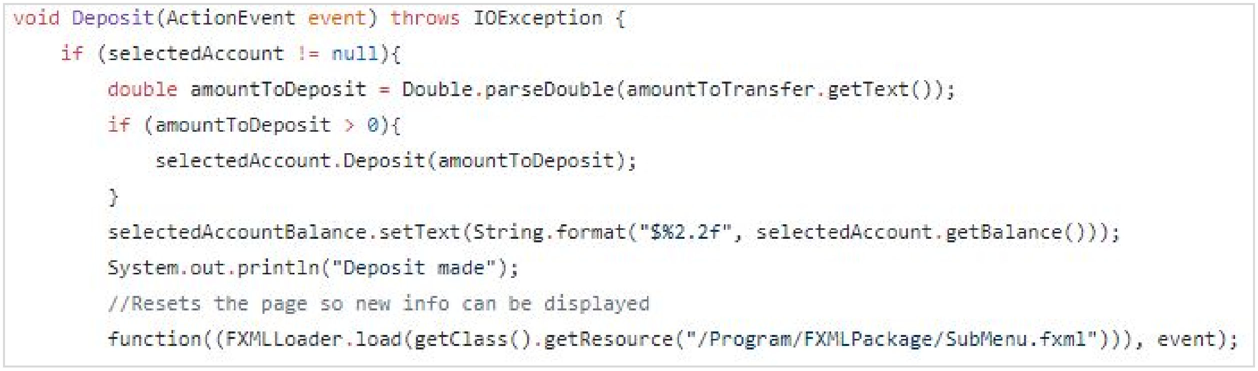
```
void returnTo(ActionEvent event) throws IOException {
    if (Main.currentAuthorization != null) {
        if (Main.currentAuthorization == EnumeratedTypes.TELLER)
            function((FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/TellerMainMenu.fxml"))), event);
        else if (Main.currentAuthorization == EnumeratedTypes.MANAGER)
            function((FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/ManagerMainMenu.fxml"))), event);
```

3.2.7 closeAccount() - This used for close account.



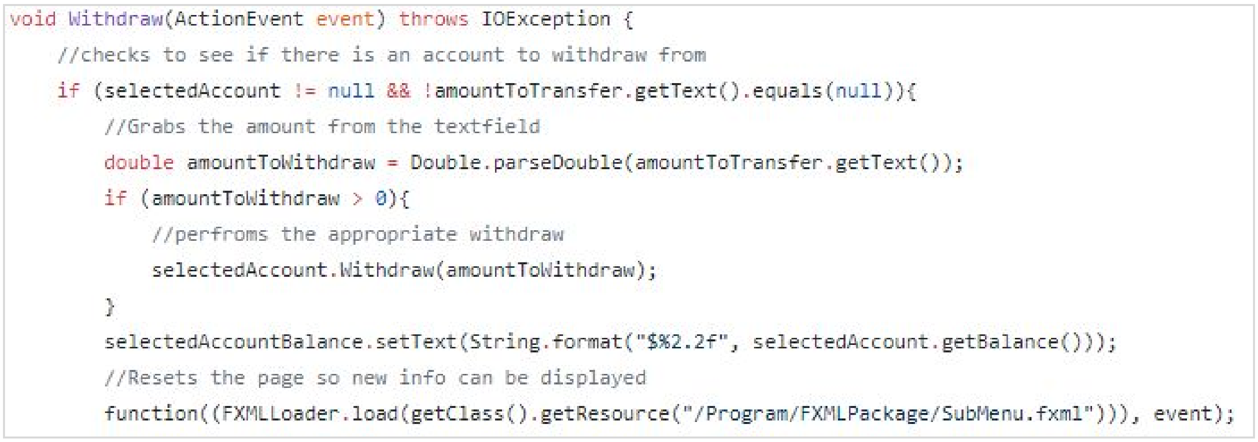
```
void closeAccount(ActionEvent event) throws IOException
{
    if (selectedAccount3 != null){
        double amountToGive = selectedAccount3.CloseAccount(LocalDate.now());
        System.out.println(String.format("%2.2f", amountToGive));
        manCusSelect.getItems().remove(selectedAccount3);
        function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/SubMenu.fxml")), event);
```

3.2.8 Deposit() - This used for deposit the money to account.



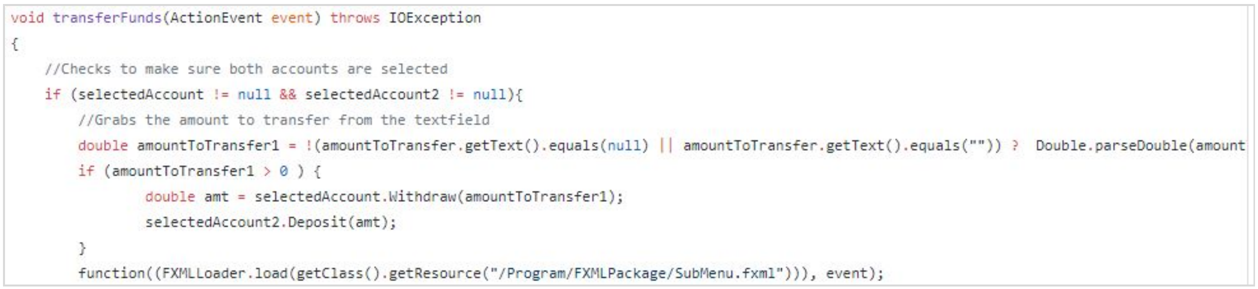
```
void Deposit(ActionEvent event) throws IOException {
    if (selectedAccount != null){
        double amountToDeposit = Double.parseDouble(amountToTransfer.getText());
        if (amountToDeposit > 0){
            selectedAccount.Deposit(amountToDeposit);
        }
        selectedAccountBalance.setText(String.format("$%2.2f", selectedAccount.getBalance()));
        System.out.println("Deposit made");
        //Resets the page so new info can be displayed
        function((FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/SubMenu.fxml"))), event);
```

3.2.9 Withdraw() - This used for withdraw money from account.



```
void Withdraw(ActionEvent event) throws IOException {
    //checks to see if there is an account to withdraw from
    if (selectedAccount != null && !amountToTransfer.getText().equals(null)){
        //Grabs the amount from the textfield
        double amountToWithdraw = Double.parseDouble(amountToTransfer.getText());
        if (amountToWithdraw > 0){
            //perfroms the appropriate withdraw
            selectedAccount.Withdraw(amountToWithdraw);
        }
        selectedAccountBalance.setText(String.format("$%2.2f", selectedAccount.getBalance()));
        //Resets the page so new info can be displayed
        function((FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/SubMenu.fxml"))), event);
```

3.2.10 transferFunds() - This used for checking or saving account transfer funds to other.

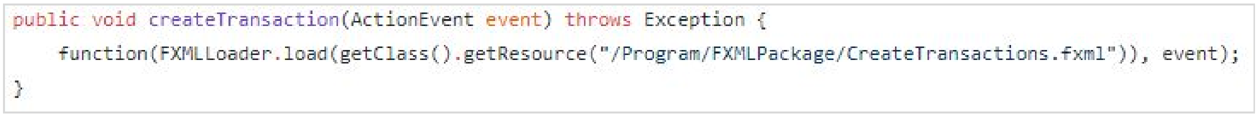


```
void transferFunds(ActionEvent event) throws IOException
{
    //Checks to make sure both accounts are selected
    if (selectedAccount != null && selectedAccount2 != null){
        //Grabs the amount to transfer from the textfield
        double amountToTransfer1 = !(amountToTransfer.getText().equals(null) || amountToTransfer.getText().equals("")) ?  Double.parseDouble(amount
        if (amountToTransfer1 > 0 ) {
                double amt = selectedAccount.Withdraw(amountToTransfer1);
                selectedAccount2.Deposit(amt);
        }
        function((FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/SubMenu.fxml"))), event);
```

3.2.11 initialize() - Sets customers account being displayed to the selected customer on the Manager/Teller page. Within the function, there are various controllers that change what is being displayed.

3.2.12 changed() - Depending on the combobox selected, it will change the information in the fields being displayed

3.2.13 createTranscation() - This used for user create a transaction.

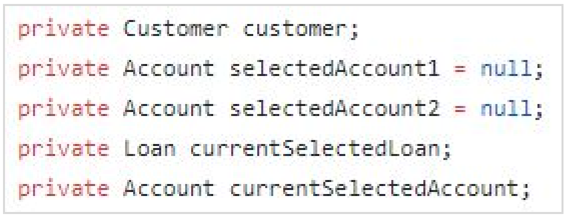


```
public void createTransaction(ActionEvent event) throws Exception {
    function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/CreateTransactions.fxml")), event);
}
```

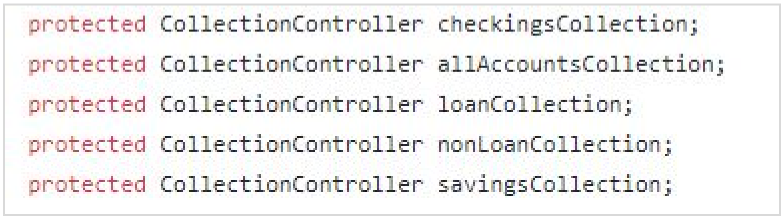
3.3 CustomerController.java

The CustomerController handles the functions a customer is allowed to perform while logged in.

3.3.1 Variables - In addition to the below image there are many Button, ComboBox, TextField, TextArea and ListView variables as well.

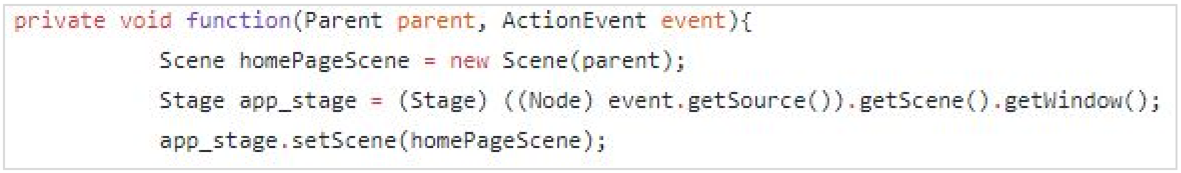


```
private Customer customer;
private Account selectedAccount1 = null;
private Account selectedAccount2 = null;
private Loan currentSelectedLoan;
private Account currentSelectedAccount;
```



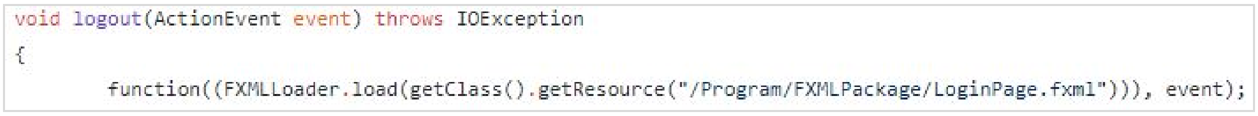
```
protected CollectionController checkingsCollection;
protected CollectionController allAccountsCollection;
protected CollectionController loanCollection;
protected CollectionController nonLoanCollection;
protected CollectionController savingsCollection;
```

3.3.2 function() - This used for changing the gui pages and setting the new scene.



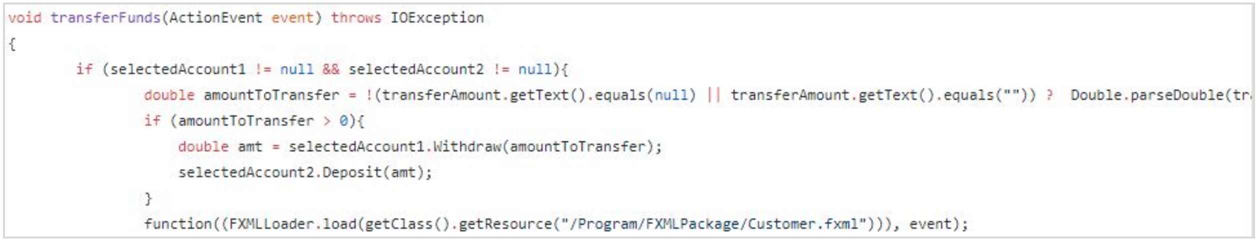
```
private void function(Parent parent, ActionEvent event){
            Scene homePageScene = new Scene(parent);
            Stage app_stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
            app_stage.setScene(homePageScene);
```

3.3.3 logout() - This used for user log out the page.



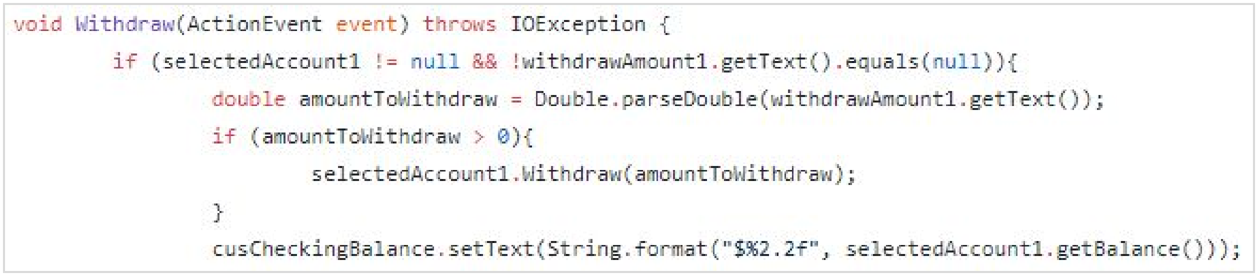
```
void logout(ActionEvent event) throws IOException
{
        function((FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/LoginPage.fxml"))), event);
```

3.3.4 transferFunds() - This used for checking or saving account transfer funds to other.



```
void transferFunds(ActionEvent event) throws IOException
{
        if (selectedAccount1 != null && selectedAccount2 != null){
                double amountToTransfer = !(transferAmount.getText().equals(null) || transferAmount.getText().equals("")) ?  Double.parseDouble(tr
                if (amountToTransfer > 0){
                    double amt = selectedAccount1.Withdraw(amountToTransfer);
                    selectedAccount2.Deposit(amt);
                }
                function((FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/Customer.fxml"))), event);
```

3.3.5 Withdraw() - This used for withdraw money from account.

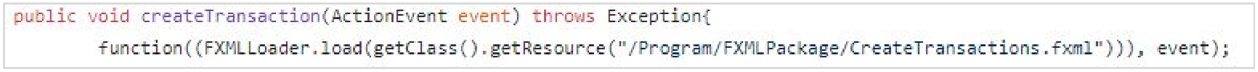


```
void Withdraw(ActionEvent event) throws IOException {
        if (selectedAccount1 != null && !withdrawAmount1.getText().equals(null)){
                double amountToWithdraw = Double.parseDouble(withdrawAmount1.getText());
                if (amountToWithdraw > 0){
                        selectedAccount1.Withdraw(amountToWithdraw);
                }
                cusCheckingBalance.setText(String.format("$%2.2f", selectedAccount1.getBalance()));
```

3.3.6 initialize() - updating display to show customers information.

3.3.7 changed() setting local variables.

3.3.8 createTranscation() - This used for user create transaction.

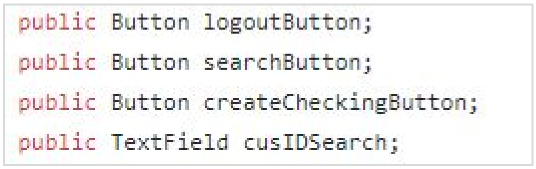


```
public void createTransaction(ActionEvent event) throws Exception{
        function((FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/CreateTransactions.fxml"))), event);
```

3.4 TellerController.java

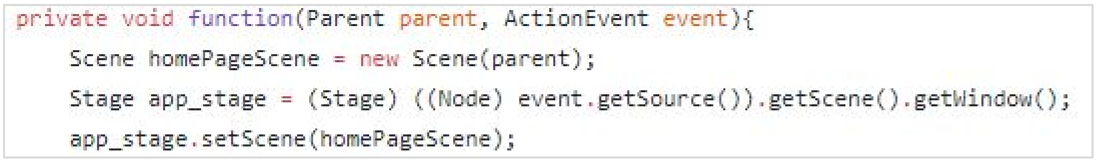
The TellerController handles the functions a teller is allowed to perform while logged in.

3.4.1 Variables.



```
public Button logoutButton;
public Button searchButton;
public Button createCheckingButton;
public TextField cusIDSearch;
```

3.4.2 function()



```
private void function(Parent parent, ActionEvent event){
    Scene homePageScene = new Scene(parent);
    Stage app_stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
    app_stage.setScene(homePageScene);
```

3.4.3 logout() - Allows a teller to log out.

```
void logout(ActionEvent event) throws IOException
{
    function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/LoginPage.fxml")), event);
}
```

3.4.4 telSearch() - Method for a teller to search for a customer in the system by SSN.

```
void telSearch(ActionEvent event) throws IOException
{
    String ssn = !cusIDSearch.getText().isEmpty() ? cusIDSearch.getText() : "";
    if (Main.findCustomer(ssn)) {
        Main.currentCustomer = Main.getCustomer(ssn);
        function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/SubMenu.fxml")), event);
```

3.4.5 createAccount() - Method to begin creating an account from the teller page.

```
void createAccount(ActionEvent event) throws IOException
{
    function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/AccountCreation.fxml")), event);
```

3.4.6 initialize(`URL arg0, ResourceBundle arg1`)

3.5 ManagerController.java

The ManagerController handles the functions a manager is allowed to perform while logged in.

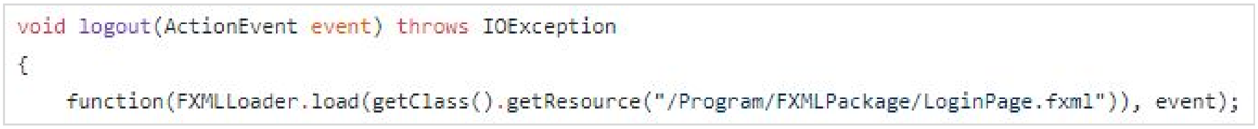
3.5.1 Variables.

```
public Button logoutButton;
public Button searchButton;
public Button createSavingsButton;
public Button createLoanButton;

public TextField cusIDSearch;
```

3.5.2 function()

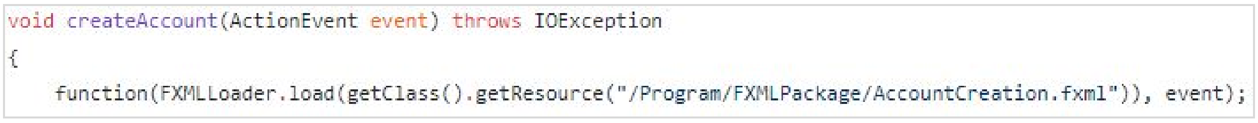
```
private void function(Parent parent, ActionEvent event){
    Scene homePageScene = new Scene(parent);
    Stage app_stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
    app_stage.setScene(homePageScene);
```

3.5.3 logout() - Method to log a manager out.



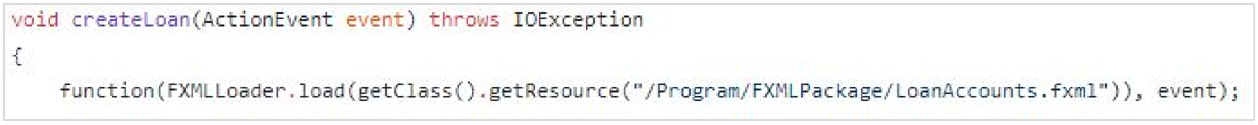
```
void logout(ActionEvent event) throws IOException
{
    function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/LoginPage.fxml")), event);
```

3.5.4 createAccount() - Method to begin creating an account for a manager.



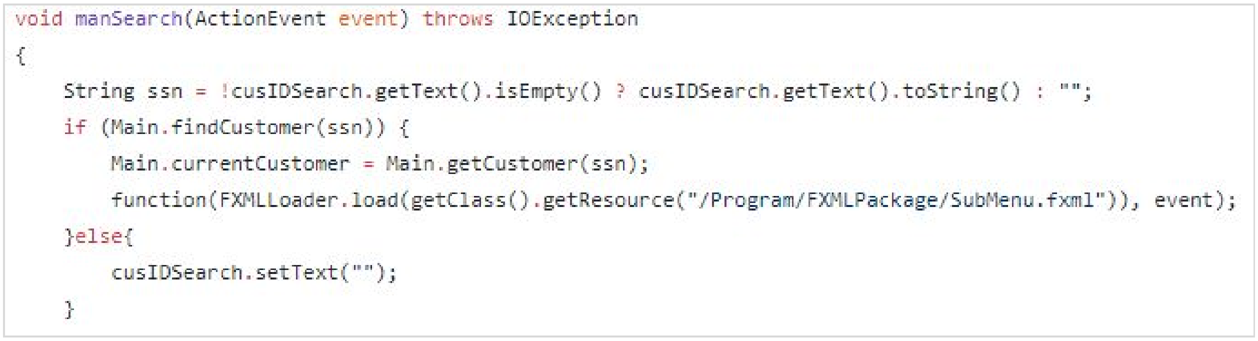
```
void createAccount(ActionEvent event) throws IOException
{
    function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/AccountCreation.fxml")), event);
```

3.5.5 createLoan() - Method to begin creating a Loan as a manager.



```
void createLoan(ActionEvent event) throws IOException
{
    function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/LoanAccounts.fxml")), event);
```

3.5.6 manSearch() - This method allows the manager to search the system for a customer.



```
void manSearch(ActionEvent event) throws IOException
{
    String ssn = !cusIDSearch.getText().isEmpty() ? cusIDSearch.getText().toString() : "";
    if (Main.findCustomer(ssn)) {
        Main.currentCustomer = Main.getCustomer(ssn);
        function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/SubMenu.fxml")), event);
    }else{
        cusIDSearch.setText("");
    }
```

3.5.7 initialize(`URL arg0, ResourceBundle arg1`)

3.6 CreateAccountController.java

The CreateAccountController handles functionality for creating a account for a customer.

3.6.1 Variables - In addition to the below image there are many Button, ComboBox, TextField, and Label variables as well.

```
ChronoLocalDate currentDate;
Customer currentCustomer;
DateFormat df = new SimpleDateFormat("mm-dd-yyyy");
String accountType;
String accountSubType;
```

3.6.2 function() - This function will be used to set and go to the next scene.

```
private void function(Parent parent, ActionEvent event) {
    Scene homePageScene = new Scene(parent);
    Stage app_stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
    app_stage.setScene(homePageScene);
}
```

3.6.3 returnTo() - This function, based on login permissions, will take you back to the manager or teller main page.

```
void returnTo(ActionEvent event) throws IOException {
    if (Main.currentAuthorization != null) {
        if (Main.currentAuthorization == EnumeratedTypes.TELLER)
            function((FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/TellerMainMenu.fxml"))), event);
        else if (Main.currentAuthorization == EnumeratedTypes.MANAGER)
            function((FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/ManagerMainMenu.fxml"))), event);
```

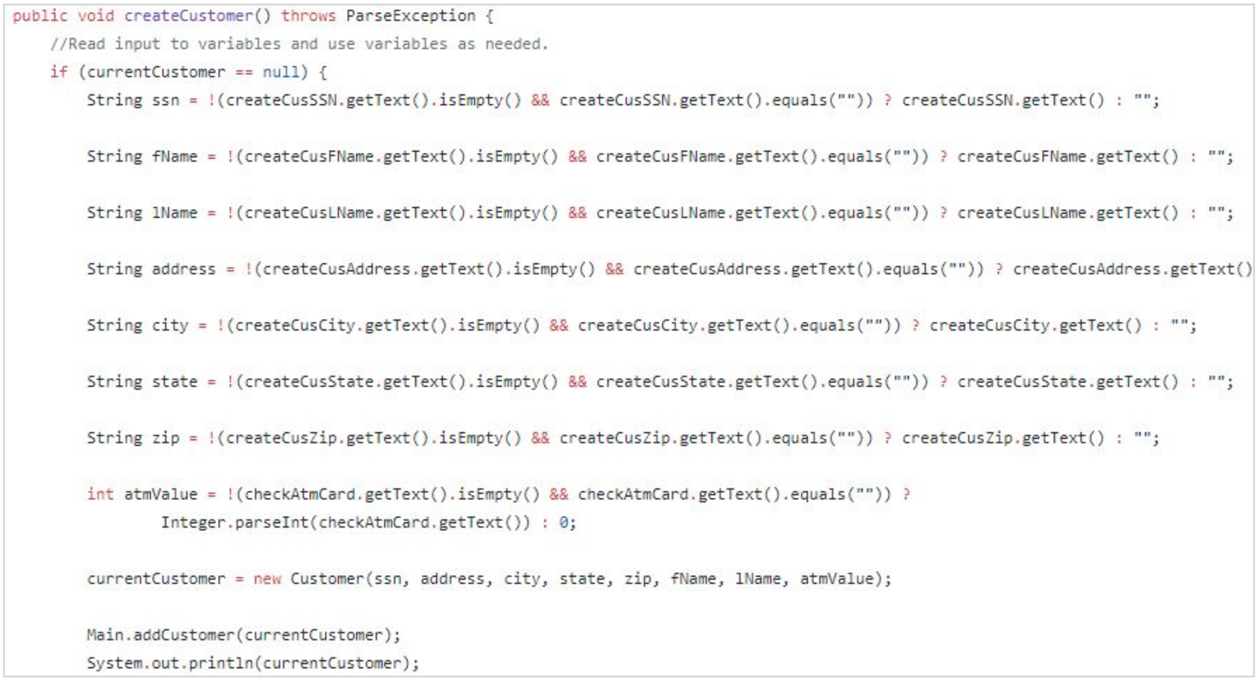
3.6.4 createAccount() - This method will create an account object as long as there is a customer to create an account for.

```
public void createAccount() throws ParseException {
    if (currentCustomer != null) {
```

Lastly, account is sent to main to be added to the accounts ArrayList.

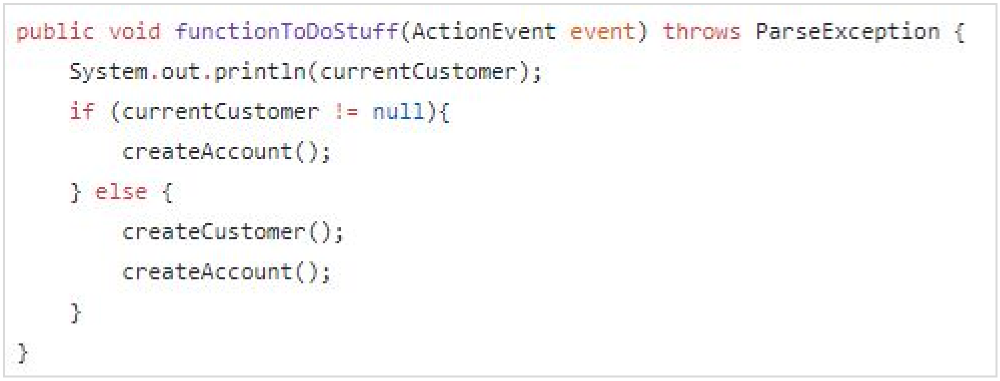
```
    }
    System.out.println(account);
    Main.addAccount(account);
```

3.6.5 createCustomer() When called it reads the input variables and creates a customer object. The customer is sent to main and added to the customer ArrayList.



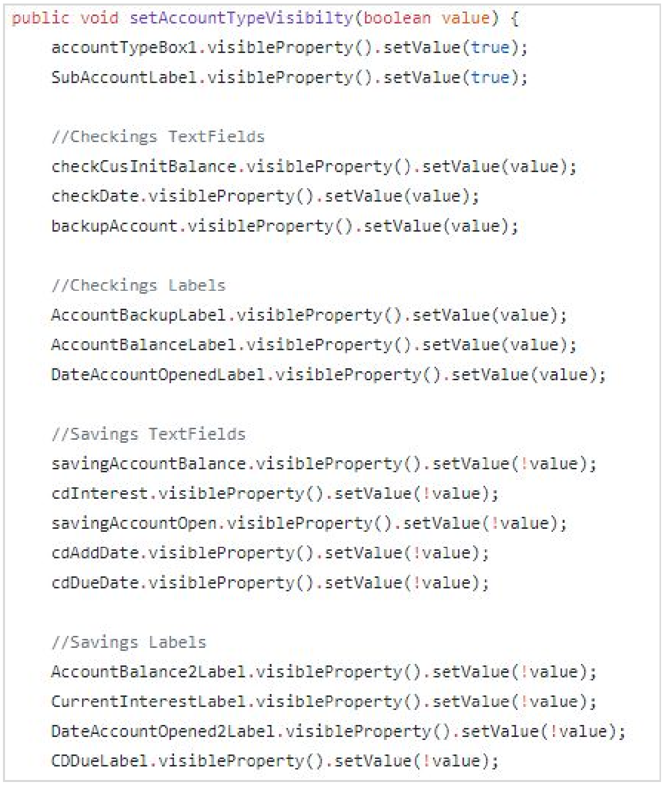
```java
public void createCustomer() throws ParseException {
    //Read input to variables and use variables as needed.
    if (currentCustomer == null) {
        String ssn = !(createCusSSN.getText().isEmpty() && createCusSSN.getText().equals("")) ? createCusSSN.getText() : "";

        String fName = !(createCusFName.getText().isEmpty() && createCusFName.getText().equals("")) ? createCusFName.getText() : "";

        String lName = !(createCusLName.getText().isEmpty() && createCusLName.getText().equals("")) ? createCusLName.getText() : "";

        String address = !(createCusAddress.getText().isEmpty() && createCusAddress.getText().equals("")) ? createCusAddress.getText()

        String city = !(createCusCity.getText().isEmpty() && createCusCity.getText().equals("")) ? createCusCity.getText() : "";

        String state = !(createCusState.getText().isEmpty() && createCusState.getText().equals("")) ? createCusState.getText() : "";

        String zip = !(createCusZip.getText().isEmpty() && createCusZip.getText().equals("")) ? createCusZip.getText() : "";

        int atmValue = !(checkAtmCard.getText().isEmpty() && checkAtmCard.getText().equals("")) ?
                Integer.parseInt(checkAtmCard.getText()) : 0;

        currentCustomer = new Customer(ssn, address, city, state, zip, fName, lName, atmValue);

        Main.addCustomer(currentCustomer);
        System.out.println(currentCustomer);
```

3.6.6 functionToDoStuff()



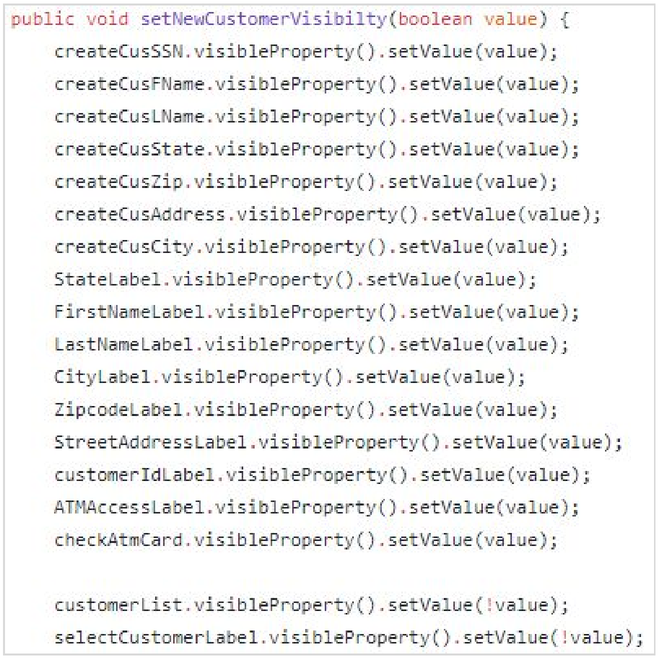
```java
public void functionToDoStuff(ActionEvent event) throws ParseException {
    System.out.println(currentCustomer);
    if (currentCustomer != null){
        createAccount();
    } else {
        createCustomer();
        createAccount();
    }
}
```

3.6.7 setAccountTypeVisibilty()



```java
public void setAccountTypeVisibilty(boolean value) {
    accountTypeBox1.visibleProperty().setValue(true);
    SubAccountLabel.visibleProperty().setValue(true);

    //Checkings TextFields
    checkCusInitBalance.visibleProperty().setValue(value);
    checkDate.visibleProperty().setValue(value);
    backupAccount.visibleProperty().setValue(value);

    //Checkings Labels
    AccountBackupLabel.visibleProperty().setValue(value);
    AccountBalanceLabel.visibleProperty().setValue(value);
    DateAccountOpenedLabel.visibleProperty().setValue(value);

    //Savings TextFields
    savingAccountBalance.visibleProperty().setValue(!value);
    cdInterest.visibleProperty().setValue(!value);
    savingAccountOpen.visibleProperty().setValue(!value);
    cdAddDate.visibleProperty().setValue(!value);
    cdDueDate.visibleProperty().setValue(!value);

    //Savings Labels
    AccountBalance2Label.visibleProperty().setValue(!value);
    CurrentInterestLabel.visibleProperty().setValue(!value);
    DateAccountOpened2Label.visibleProperty().setValue(!value);
    CDDueLabel.visibleProperty().setValue(!value);
```

3.6.8 setNewCustomerVisibilty()



```java
public void setNewCustomerVisibilty(boolean value) {
    createCusSSN.visibleProperty().setValue(value);
    createCusFName.visibleProperty().setValue(value);
    createCusLName.visibleProperty().setValue(value);
    createCusState.visibleProperty().setValue(value);
    createCusZip.visibleProperty().setValue(value);
    createCusAddress.visibleProperty().setValue(value);
    createCusCity.visibleProperty().setValue(value);
    StateLabel.visibleProperty().setValue(value);
    FirstNameLabel.visibleProperty().setValue(value);
    LastNameLabel.visibleProperty().setValue(value);
    CityLabel.visibleProperty().setValue(value);
    ZipcodeLabel.visibleProperty().setValue(value);
    StreetAddressLabel.visibleProperty().setValue(value);
    customerIdLabel.visibleProperty().setValue(value);
    ATMAccessLabel.visibleProperty().setValue(value);
    checkAtmCard.visibleProperty().setValue(value);

    customerList.visibleProperty().setValue(!value);
    selectCustomerLabel.visibleProperty().setValue(!value);
```

3.6.9 initialize() - Sets the values in the sub-menu CreateAccount to the selected account type.

3.6.10 changed() - Checks to see if you are using a new or pre existing customer. Changes the fields displayed depending on what was selected in the combo box.

3.7 LoanAccountController.java

The LoanAccountController handles functionality for creating loans in the system.

3.7.1 Variables:

```
//Variables for the FXML values
private Customer currentSelectedCustomer;

public Button returnMenu;
public Button createLoan;

public DatePicker notificationDate;
public DatePicker dueDate;

public TextField cusInterestRate;
public TextField cusLoanDesc;
public TextField cusLoanAmount;

public ComboBox loanType;
public ComboBox cusID;
```

3.7.2 function() - This is used to traverse pages.

```
private void function(Parent parent, ActionEvent event){
    //function to traverse pages
    Scene homePageScene = new Scene(parent);
    Stage app_stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
    app_stage.setScene(homePageScene);
}
```

3.7.3 createLoan() - Checks that there is not a null so a loan can be created

```
try {
    if (!currentSelectedCustomer.equals(null)) {
        String desc = !cusLoanDesc.getText().equals(null) ? cusLoanDesc.getText() : "";
        double balance = !cusLoanAmount.getText().equals(null) ? Double.parseDouble(cusLoanAmount.getText()) : 0.00;
        double interestRate = !cusInterestRate.getText().equals(null) ? Double.parseDouble(cusInterestRate.getText()) : 0.00;
```

Grabs the dates.

```
LocalDate dueOn1 = dueDate.getValue() != null ? dueDate.getValue() : LocalDate.now();
LocalDate notifyOn1 = notificationDate !=null ? notificationDate.getValue(): LocalDate.now();
Calendar c1 = Calendar.getInstance();
Calendar c2 = Calendar.getInstance();
c1.set(dueOn1.getYear(),dueOn1.getMonthValue()-1,dueOn1.getDayOfMonth());
c2.set(notifyOn1.getYear(),notifyOn1.getMonthValue()-1,notifyOn1.getDayOfMonth());

ChronoLocalDate dueOn = LocalDate.of(c1.get(Calendar.YEAR), c1.get(Calendar.MONTH) + 1, c1.get(Calendar.DAY_OF_MONTH));
ChronoLocalDate notifyOn = LocalDate.of(c2.get(Calendar.YEAR), c2.get(Calendar.MONTH) + 1, c2.get(Calendar.DAY_OF_MONTH));
ChronoLocalDate today = LocalDate.of(Calendar.getInstance().get(Calendar.YEAR), Calendar.getInstance().get(Calendar.MONTH)
```

The following show where the loan type string is abbreviated so it can be passed to the constructor

```
String loanTypeShort = (String) loanType.getValue();
if (loanTypeShort.equals("Credit Card")){loanTypeShort="CC";}
if (loanTypeShort.equals("Long Term Loan")){loanTypeShort="LT";}
if (loanTypeShort.equals("Short Term Loan")){loanTypeShort="ST";}
```

The loan is then created and added to the accounts ArrayList in Main.

```
Loan newLoan = new Loan(currentSelectedCustomer.getCustomerId(),0, desc, balance, interestRate, dueOn, notifyOn,0,dueOn,false, loanTypeShort);
//adds the new loan to the database
Main.addAccount(newLoan);
```

3.7.4 returnMan() - Returns the GUI to the manager main menu screen.

```
void returnMan(ActionEvent event) throws IOException
{
    function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/ManagerMainMenu.fxml")), event);
}
```

3.7.5 initialize()

```
public void initialize(URL location, ResourceBundle resources) {
    cusID.setItems((new CollectionController(Main.customers).getCollections()));

    cusID.valueProperty().addListener(new ChangeListener() {
        @Override
        public void changed(ObservableValue observable, Object oldValue, Object newValue) {
            currentSelectedCustomer = (Customer) newValue;
        }
    });
}
```

3.8 CollectionController.java

Taking in an ArrayList and turning them into a ObservableList for use with combo boxes.

3.8.1 CollectionController{}.

```
public class CollectionController {
    private ObservableList<?> collections;

    public CollectionController(ArrayList newCollection){
        collections = FXCollections.observableArrayList(newCollection);
    }

    public ObservableList<?> getCollections() {
        return collections;
    }
}
```

3.9 TransactionController.java

The TransactionController is used for obtaining values from the CreateTransactions.fxml page. The controller should pass the content to savings the account class.

3.9.1 Variables.

```
public ComboBox selectAccountType;
public ComboBox selectCustomer;

// Credit card input variables
public TextField creditName;
public TextField creditAmount;
public Button submitCredit;

// Checking input variables
public TextField checkName;
public TextField checkAmount;
public Button submitCheck;
```

3.9.2 function() - This used for changing the gui pages and setting the new scene.

```
private void function(Parent parent, ActionEvent event) {
    //function to traverse pages
    Scene homePageScene = new Scene(parent);
    Stage app_stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
    app_stage.setScene(homePageScene);
}
```

3.9.3 returnCheckTransaction(`ActionEvent actionEvent`) - Used to submit the credit card transaction.

3.9.4 returnCheckTransaction(ActionEvent actionEvent) - Used to submit the checking transactions.

3.9.5 returnToCus() - Checks the login type and makes sure to transfer the user to the correct area when navigating.

```
void returnToCus(ActionEvent event) throws IOException {

    if (Main.currentAuthorization != null) {

        if (Main.currentAuthorization == EnumeratedTypes.TELLER)
            function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/Submenu.fxml")), event);

        else if (Main.currentAuthorization == EnumeratedTypes.MANAGER)
            function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/SubMenu.fxml")), event);

        else if (Main.currentAuthorization == EnumeratedTypes.CUSTOMER)
            function(FXMLLoader.load(getClass().getResource("/Program/FXMLPackage/Customer.fxml")), event);
    }
}
```

4. Data

4.1 BankingDatabase.txt - This text file is the main data file for the banking system. It is accessed only at the startup and shutdown of the system. Each block of data in the file is headed by the type of data to follow. Examples to follow:

```
Customer Account
423453245,114 North 4th,Clarksdale,MO,64493,Ronald,Jones,1
345653425,1805 Jules,St. Joseph,MO,64503,Mark,Ingrem,1
```

```
345679898,703 Park Ln,St. Joseph,MO,64501,Broderick,Jones,0

Savings Account
423453245,458097515,66073.00,1.90,06-18-2017,04-19-2019
345653425,1931709156,651.00,1.90,07-31-2017,11-07-2018
```

```
563432913,-1107009920,54654.00,1.90,10-09-2017,

Checking Account
423453245,1138040035,3192.98,regular,458097515,0,01-30-2008
345653425,-1184285048,354.03,regular,0,0,02-20-2002
```

```
423453245,-53255176,900.00,gold,0,0,11-30-2018

Loan Account
423453245,833783780,ShortLoan,638.00,0.056,01-27-2018,09-25-2018,99.95,09-01-2018,0,ST
423453245,812136543,CreditCard,600.09,0.056,10-30-2018,10-01-2018,66.81,09-01-2018,0,CC
```

```
345812332,-1158685762,House,150000.00,0.045,01-14-2019,11-14-2018,562.50,12-15-2018,0,LT

Transactions Account
-1553056732,-2032970349,15.50,Dick's Sporting Goods,11-22-2018,0
1224090941,-2032970349,25.75,Newegg.com,11-22-2018,0
```

4.2 EnumeratedTypes.java - See entire file below:

```java
package Program.Data;

public enum EnumeratedTypes {CUSTOMER, TELLER, MANAGER}
```

5. FXMLPackage

In the FXML package are the generated FXML files used for the GUI objects and styling. The following is the list of all the files:

5.1 AccountCreation.fxml
5.2 CreateTransactions.fxml
5.3 Customer.fxml
5.4 Failed.fxml
5.5 LoanAccounts.fxml
5.6 LoginPage.fxml
5.7 ManagerMainMenu.fxml
5.8 SavingsAccount.fxml
5.9 SubMenu.fxml
5.10 Success.fxml
5.11 TellerMainMenu.fxml