Java Final Project Documentation

Zhenghan PEI, Mathurin MAUSSION, Amine LAKZIZ

January 17, 2024

Abstract

The financial management application discussed herein presents a Java-based solution that empowers users with a versatile platform for comprehensive financial interactions. Encompassing features ranging from user authentication and account creation to investment management, cryptocurrency transactions, fund transfers, and loan applications, the system integrates seamlessly with a C++ server. Leveraging JavaFX for a dynamic graphical user interface, the application employs modular controllers, utility classes, and robust data structures. This abstract encapsulates a holistic approach towards providing users with an intuitive and efficient financial management experience.

Documentation

The documentation for this project has been authored by **Zhenghan PEI**.

Navigating the Documentation

To enhance user experience, the documentation has been structured with a user-friendly menu system. Each section and its corresponding subtopics have been meticulously organized to facilitate quick and efficient access to specific information.

Menu Structure

1 Data Structures

1.1 Investment Class

The `Investment` class represents the investment profile of a bank account, encompassing the number of cryptocurrencies owned (`coins`) and a map (`investMap`) detailing the amount of stocks owned for different companies.

1.1.1 Class Fields

- **coins:** `float` - Represents the number of cryptocurrencies owned by the bank account.
- **investMap:** `Map<String, Float>` - A map associating the names of companies with the amount of stocks owned for each.

1.1.2 Constructor

The class has a constructor that initializes the `investMap` and `coins` based on the provided `investInfos` string.

```
public Investment(String investInfos) {
    investMap = new HashMap<>();
    coins = 0.0F;
    if (investInfos.isEmpty()) {
        return;
    }
    String[] investments = investInfos.split(",");

    for (String investment : investments) {
        String[] parts = investment.trim().split(" ");
        String key = parts[0];
        float value = Float.parseFloat(parts[1]);
        if (key.equals("coin")) {
            coins = value;
        } else {
            investMap.put(key, value);
        }
    }
}
```

1.1.3 `toString` Method

The class overrides the `toString` method to provide a formatted string representation of the investment, including the number of coins and details about the financial products.

```
@Override
public String toString() {
    StringBuilder res = new StringBuilder("Coins: " + coins + "\n" +
            "Financial Products: ");
    for (Map.Entry<String, Float> entry : investMap.entrySet()) {
        String key = entry.getKey();
        Float value = entry.getValue();

        res.append("\n\t").append(key).append(": ").append(value);
    }
    return res.toString();
}
```

The `Investment` class encapsulates information about cryptocurrency and stock ownership, providing a convenient way to manage and represent investment profiles associated with bank accounts. The constructor and `toString` method offer flexibility in initializing the class and obtaining a human-readable representation of the investment details.

1.2 BankAccount Class

The `BankAccount` class represents an individual's bank account and includes information about currency, deposit, debt, and associated investments.

1.2.1 Class Fields

- **currency:** `float` - Represents the amount of currency held in the bank account.
- **deposit:** `float` - Represents the deposited amount in the bank account.
- **debt:** `float` - Represents the debt amount associated with the bank account.
- **investment:** `Investment` - Represents the investment profile associated with the bank account.

1.2.2 Constructor

The class has a constructor that initializes the fields based on the provided `infos` string.

```
public BankAccount(String infos) {
    // Constructor details
}
```

1.2.3 `toString` Method

The class overrides the `toString` method to provide a formatted string representation of the bank account.

```
@Override
public String toString() {
    // toString method details
}
```

1.3 UserObj Class

The `UserObj` class represents user information and includes static fields for a single logged-in user. It encapsulates details such as the username, first name, family name, telephone, email, current bank account information (`account`), and historical bank account information (`accountMap`).

1.3.1 Class Fields

- **username:** `String` - Represents the username of the logged-in user.
- **first_name:** `String` - Represents the first name of the logged-in user.
- **family_name:** `String` - Represents the family name of the logged-in user.
- **telephone:** `String` - Represents the telephone number of the logged-in user.
- **e_mail:** `String` - Represents the email address of the logged-in user.
- **account:** `BankAccount` - Represents the bank account information of the logged-in user.
- **accountMap:** `Map<String, BankAccount>` - A map containing the bank account information of the user by date.

1.3.2 `toStringUserObj` Method

The class includes a method, `toStringUserObj`, to provide a formatted string representation of the user object, including details such as username, first name, family name, telephone, email, current bank account information, and historical bank account information.

```
public static String toStringUserObj(){
    StringBuilder result = new StringBuilder();
    result.append("Username: ").append(username).append('\n');
    result.append("First Name: ").append(first_name).append('\n');
    result.append("Family Name: ").append(family_name).append('\n');
    result.append("Telephone: ").append(telephone).append('\n');
    result.append("E-mail: ").append(e_mail).append('\n');

    if (account != null) {
        result.append("Account: ").append(account).append('\n');
    } else {
        result.append("No account information available").append('\n');
    }

    if (!accountMap.isEmpty()) {
        result.append("Account Map: ").append('\n');
        for (Map.Entry<String, BankAccount> entry : accountMap.entrySet()) {
            result.append(entry.getKey()).append(": ").append(entry.getValue()).append('\n');
        }
    } else {
        result.append("No account map information available").append('\n');
    }

    return result.toString();
}
```

The `UserObj` class serves as a centralized structure for managing user information, including details about the logged-in user, their current bank account, and historical bank account information. The `toStringUserObj` method provides a user-friendly representation of the user object.

1.4 GlobalObj Class

The `GlobalObj` class is a crucial component for managing various global information related to the application. It encompasses fields and methods to handle data such as sale and purchase of cryptocurrencies, current coin values, historical coin data, company stock values, and historical company stock data.

1.4.1 Class Fields

- **saleCoins:** `Map<String, Float>` - Represents users who want to sell cryptocurrencies and the corresponding amounts they have registered to sell.
- **buyCoins:** `Map<String, Float>` - Represents users who want to buy cryptocurrencies and the amount of money they have registered to spend.
- **coin:** `float` - Represents the current value of the cryptocurrency.
- **coinMap:** `Map<String, Float>` - A historical map that stores the values of the cryptocurrency by date.
- **currDataMap:** `Map<String, Float>` - A map containing the current stock values of each company.
- **dataMap:** `Map<String, Map<String, Float>>` - A map containing historical stock values of each company by date.

1.4.2 Static Initialization Block

The class includes a static initialization block that creates instances of the necessary maps, ensuring they are ready for use.

```
static {
    saleCoins = new HashMap<>();
    buyCoins = new HashMap<>();
    coinMap = new HashMap<>();
    currDataMap = new HashMap<>();
    dataMap = a HashMap<>();
}
```

1.4.3 Static Methods

- `initializeClass(String globInfos)`: Parses a string containing global information and populates the class fields accordingly. The string is expected to have specific parts separated by semicolons.
- `toStringGlobalObj()`: Generates a string representation of the entire class, including all maps and values.

1.4.4 Helper Method

- `appendMapToString(StringBuilder builder, Map<String, Float> map)`: Appends the entries of a map to a StringBuilder, creating a formatted string.

The `GlobalObj` class serves as a centralized repository for managing and accessing essential global data within the application. It ensures a structured and organized approach to handling information related to cryptocurrency sales, purchases, and historical stock values.

1.5 Testing Classes

1.5.1 BankAccountTest Class

The `BankAccountTest` class contains test methods to validate the initialization of the `BankAccount` class. It includes a private method to generate random float values for testing purposes.

1.5.2 GlobalObjTest Class

The `GlobalObjTest` class contains test methods to validate the initialization and string representation of the `GlobalObj` class.

1.5.3 InvestmentTest Class

The `InvestmentTest` class contains test methods to validate the initialization of the `Investment` class with and without coins. It also includes a method to compare investment maps and a private method to generate company investment information.

1.5.4 UserObjTest Class

The `UserObjTest` class contains test methods to validate the initialization of the `UserObj` class. It includes private methods to generate random usernames, names, telephone numbers, and email addresses for testing purposes.

2 Tools and Realizations

2.1 ChartGenerator Class

The `ChartGenerator` class, located in the `Tools` package, provides functionalities for generating and displaying various charts related to financial data. It utilizes JavaFX for creating line charts and Java's standard libraries for handling data.

2.1.1 Methods

- `chartAccountData()`: Generates a line chart representing the account data over time, including currency, deposit, debt, and their sum.
- `chartGlobal()`: Generates a line chart representing the historical stock values over time for various companies.
- `chartSelf()`: Generates a line chart representing the user's self-investments over time, considering both cryptocurrency and stock investments.

2.1.2 Dependencies

The class relies on the following data structures and classes:

- `BankAccount` class: Represents individual bank account data.
- `GlobalObj` class: Manages global information related to cryptocurrency sales, purchases, and stock values.
- `UserObj` class: Represents user information, including bank account details.

2.1.3 JavaFX Components

The class uses JavaFX components to create and display line charts:

- `LineChart`: Represents the main line chart.
- `CategoryAxis` and `NumberAxis`: Define the axes for the line chart.
- `XYChart.Series` and `XYChart.Data`: Represent series and data points on the chart.
- `Alert`: Displays information, warnings, and errors in a pop-up dialog.
- `VBox`: A layout component for vertical arrangement of JavaFX components.

2.1.4 Chart Generation

The methods use historical data stored in the `UserObj.accountMap` and `GlobalObj.dataMap` to create informative line charts. The charts include details such as account currency, deposit, debt, stock values, and self-investments over time.

2.1.5 Date Parsing

The class includes a method, `parseDate`, to parse date strings into Java `Date` objects, facilitating chronological sorting of data.

2.1.6 Chart Display

The generated charts are displayed using JavaFX `Alert` dialogs, providing a user-friendly way to visualize financial data.

2.1.7 Series Management

The class efficiently manages series creation and addition to the line charts, ensuring a clear representation of different data categories.

2.1.8 Chart Limitations

While the class provides valuable insights into financial data trends, it has a limitation in displaying a maximum of 30 entries on the charts to prevent clutter and maintain readability.

2.2 JavaClient Class

The `JavaClient` class, located in the `Tools` package, facilitates communication with a server through a socket connection. It is designed to send commands to the server and receive corresponding responses. The class is crucial for integrating the application with a server written in C++ that constantly listens for connections on the specified socket.

2.2.1 Socket Communication Setup

The class establishes a socket connection to the server using the address "localhost" and port 12345. It includes components for writing output to the server (`PrintWriter`) and reading responses from the server (`BufferedReader`). Additionally, a separate thread (`responseThread`) is created to continuously monitor and process server responses.

2.2.2 Threaded Response Handling

The `responseThread` reads server responses in a loop and places them in a `BlockingQueue` (`responseQueue`). This allows the main program to asynchronously retrieve responses from the server without blocking the execution flow. The thread continues to run until the socket is closed, ensuring constant monitoring for incoming responses.

2.2.3 Command Sending and Response Retrieval

The class provides methods for sending commands to the server. The `sendAndReceive` method sends a command to the server and waits for the corresponding response. This is achieved by utilizing the `BlockingQueue`, where the response thread deposits server responses. The `send` method allows sending commands without waiting for a response, providing flexibility in communication.

2.2.4 Socket Closure

The `close` method is responsible for closing the socket and associated resources. It ensures a clean termination by closing the output writer, response reader, and the socket itself. Additionally, it waits for the `responseThread` to finish before completing the closure process.

2.2.5 Exception Handling

The class includes exception handling for potential errors during socket setup, command sending, and socket closure. Exceptions such as `IOException`, `SocketException`, and `InterruptedException` are caught and printed to the standard error stream (`e.printStackTrace()`).

2.2.6 Integration with Data Classes

The `JavaClient` class plays a pivotal role in integrating the Java application with a server, allowing the exchange of commands and responses. The responses received from the server, represented as strings, are utilized to initialize or update data classes within the Java application. This enables seamless communication and synchronization between the client-side application and the server.

2.2.7 Considerations

While the class effectively handles socket communication, it assumes a predefined protocol between the client and server, where commands and responses are exchanged in a specific format. Additionally, the class does not implement advanced error recovery mechanisms and relies on exception handling to manage potential issues during communication.

Admin Class

The `Admin` class, located in the `Tools` package, serves as a command-line interface for testing and debugging purposes during application development. This class enables manual interaction with the server by allowing the input of commands directly through the console.

Initialization

The `main` method initializes a `JavaClient` instance, establishing a connection to the server. It then enters a loop to continuously receive user input and send corresponding commands to the server. The loop continues until the user enters the command 'exit,' at which point the application terminates.

User Input Handling

The class utilizes a `Scanner` object to read user input from the console. Users can enter commands directly, providing a flexible and direct means of communication with the server.

Command Execution

User input is processed within a try-catch block, catching exceptions related to connection issues. If the user enters 'exit,' the loop breaks, and the application exits. For commands 'read' and 'save,' the `send` method of `JavaClient` is used, as these commands do not require a response. For other commands, the `sendAndReceive` method is employed to send the command and await the server's response.

2.3 Password2Hash Class

The `Password2Hash` class, located in the `Tools` package, provides a simple hashing method for passwords. The hashing algorithm involves converting each character of the password to its ASCII value plus one, multiplying these values together, and applying modular arithmetic with a predefined modulo value.

2.3.1 Hashing Algorithm

The `hashPassword` method takes a password as input and performs the following steps:
1. Convert each character of the password to its ASCII value plus one.
2. Multiply all the ASCII values together, applying modular arithmetic with a predefined modulo value (`Modulo`).
3. Return the resulting hash as an integer.

2.3.2 Modulo Value

The modulo value is set to the hexadecimal value "BAE2001." This choice provides a level of customization and uniqueness for the hashing algorithm. It is derived from the family name and birth year of the creator Zhenghan PEI (Jeonghan BAE).

2.3.3 Example Usage

The `main` method includes an example usage of the `hashPassword` method, demonstrating the calculation of a hash for the password "John Doe." The resulting hash is then printed to the console.

2.4 ShowAlert Class

The `ShowAlert` class, situated in the `Tools` package, serves as a utility for displaying different types of JavaFX alerts (`INFORMATION`, `WARNING`, and `ERROR`). This class encapsulates the creation and display of alerts, enhancing code reusability within the JavaFX program.

2.4.1 Methods

- `showAlert(String title, String message, char type):` Private method used for creating and displaying alerts based on the specified title, message, and alert type.
  - `title:` The title of the alert.
  - `message:` The content message of the alert.
  - `type:` A character representing the type of alert (`'I'` for `INFORMATION`, `'W'` for `WARNING`, `'E'` for `ERROR`).
- `Information(String title, String message):` Public method for displaying an `INFORMATION` alert.
  - `title:` The title of the alert.
  - `message:` The content message of the alert.
- `Warning(String title, String message):` Public method for displaying a `WARNING` alert.
  - `title:` The title of the alert.
  - `message:` The content message of the alert.
- `Error(String title, String message):` Public method for displaying an `ERROR` alert.
  - `title:` The title of the alert.
  - `message:` The content message of the alert.

2.4.2 Alert Types

The `ShowAlert` class provides three main types of alerts, each corresponding to a different JavaFX `AlertType`:

- `INFORMATION:` Used for informative messages.
- `WARNING:` Used for warning messages.
- `ERROR:` Used for error messages.

The alert methods allow for the specification of the title and content message, providing flexibility in alert customization.

2.5 SceneNavigator Class

The `SceneNavigator` class, situated in the `com.example.partjava` package, acts as a utility for navigating between different scenes in a JavaFX application. This class provides methods for opening new scenes and navigating to specific scenes while managing the lifecycle of stages.

2.5.1 Methods

- `openNewScene(String fxmlFileName, String title):` Opens a new scene based on the provided FXML file name and title without turning off the current scene.
  - `fxmlFileName:` The name of the FXML file representing the new scene.
  - `title:` The title of the new scene.

- `getToInterface(String fxmlFileName, Button sourceButton, String title)`: Navigates to a specific FXML file, attaches the corresponding button, and turns off the current scene. Allows customization of the title for the new scene.
  - `fxmlFileName`: The name of the FXML file representing the target scene.
  - `sourceButton`: The button associated with the current scene.
  - `title`: The title of the new scene.
- `getToInterface(String fxmlFileName, Button sourceButton)`: Default method that navigates to a specific FXML file, attaches the corresponding button, and turns off the current scene. Uses the user's full name as the title for the new scene.
  - `fxmlFileName`: The name of the FXML file representing the target scene.
  - `sourceButton`: The button associated with the current scene.

2.5.2 Lifecycle Management

The class keeps track of open stages using a `List<Stage>` named `openStages`. The `closeAllStagesExcept` method is responsible for closing all open stages except the target stage, ensuring smooth navigation between scenes. It also provides a mechanism to clear the list of open stages after closing them.

2.5.3 Usage

The `SceneNavigator` class is designed to simplify the process of navigating between scenes in a JavaFX application. It integrates with FXML files, buttons, and stages to provide a seamless and organized user interface experience.

3 Controllers

3.1 LoginController Class

The `LoginController` class, located in the `com.example.partjava` package, serves as the controller for the initial login interface in the program. This controller handles user authentication, retrieves user and bank information, updates the application's data structures, and navigates to the main user interface upon successful login.

3.1.1 FXML Elements

- `connectButton:` Button for initiating the login process.
- `newAccButton:` Button for creating a new account.
- `alertLabel:` Label for displaying alerts or messages.
- `usrNameField:` Text field for entering the username.
- `pwField:` Password field for entering the user's password.

3.1.2 Methods

- `connexionClick():` Method triggered by clicking the connectButton. It handles the login process, communicates with the server, and navigates to the main user interface upon successful authentication.
- `UpdateAccInfo(String accInfo):` Static method responsible for connecting to the server, updating the user's bank information, and populating the `UserObj.accountMap`.
- `createNewAcc():` Method triggered by clicking the newAccButton. It navigates to the NewAccount.fxml interface for creating a new account.

3.1.3 Login Process

The `connexionClick()` method performs the following steps:

1. Retrieves the entered username and password from the corresponding text fields.
2. Constructs login and bank information commands using the entered credentials.
3. Initiates a `JavaClient` to communicate with the server.
4. Sends the login command and receives user information.
5. Validates the received user information; displays an alert for invalid credentials.
6. Parses and sets user information in the `UserObj` class.
7. Sends a bank information command, updates account information, and closes the connection.
8. Retrieves global information, initializes `GlobalObj`, and prints user and global information.
9. Navigates to the main user interface using `SceneNavigator`.

3.1.4 Usage

The `LoginController` class integrates with JavaFX FXML elements, server communication (`JavaClient`), and data structures (`UserObj`, `GlobalObj`). It provides the necessary logic for user authentication and serves as the entry point to the application.

3.2 NewAccountController Class

The `NewAccountController` class, part of the `com.example.partjava` package, serves as the controller for the NewAccount.fxml interface. This controller handles the creation of a new user account, validates input fields, communicates with the server, and navigates to the login interface upon successful account creation.

3.2.1 FXML Elements

- `civilityField:` ChoiceBox for selecting the user's civility (Monsieur, Madame, `--`).
- `usernameField:` TextField for entering the desired username.
- `firstNameField:` TextField for entering the user's first name.
- `familyNameField:` TextField for entering the user's family name.
- `telephoneField:` TextField for entering the user's telephone number.
- `emailField:` TextField for entering the user's email address.
- `passwordField:` PasswordField for entering the user's password.
- `repeatPasswordField:` PasswordField for re-entering the user's password for confirmation.
- `createButton:` Button for initiating the account creation process.
- `clearButton:` Button for clearing all input fields.

3.2.2 Methods

- `initialize():` Initializes the `civilityField` with predefined choices.
- `onCreateButtonClicked():` Validates input fields, hashes the password, sends registration information to the server, and navigates to the login interface upon successful account creation.
- `onClearButtonClicked():` Clears all input fields.

3.2.3 Account Creation Process

The `onCreateButtonClicked()` method performs the following steps:

1. Validates the entered user information and displays appropriate alerts for any errors.
2. Constructs a registration information string using the entered data.
3. Initiates a `JavaClient` to communicate with the server.
4. Sends the registration information to the server and receives a response.
5. Closes the connection with the server.
6. Displays an information alert for a successful account creation and navigates to the login interface.
7. Displays a warning alert if the username already exists.

3.2.4 Usage

The `NewAccountController` class integrates with JavaFX FXML elements, server communication (`JavaClient`), and password hashing (`Password2Hash`). It provides the logic for creating a new user account and ensures data integrity through input validation.

3.3 UsersInterfaceController Class

The `UsersInterfaceController` class, part of the `com.example.partjava` package, serves as the controller for the UsersInterface.fxml interface. This controller manages the main graphical user interface, displaying user information and providing navigation to various functionalities.

3.3.1 FXML Elements

- `usernameTextArea`: TextArea for displaying the user's username.
- `fullNameTextArea`: TextArea for displaying the user's full name.
- `teleTextArea`: TextArea for displaying the user's telephone number.
- `emailTextArea`: TextArea for displaying the user's email address.
- `currencyTextArea`: TextArea for displaying the user's account currency balance.
- `depositTextArea`: TextArea for displaying the user's account deposit balance.
- `debtTextArea`: TextArea for displaying the user's account debt balance.
- `investmentTextArea`: TextArea for displaying information about the user's investments.
- `investmentButton`: Button for navigating to the investment functionality.
- `cryptoButton`: Button for navigating to the cryptocurrency functionality.
- `virementButton`: Button for navigating to the money transfer functionality.
- `loanButton`: Button for navigating to the loan functionality.
- `logOutButton`: Button for logging out and navigating to the login interface.
- `myAccountButton`: Button for viewing account-related charts.

3.3.2 Methods

- `initialize()`: Initializes the user interface by updating the displayed user information.
- `onInvestmentButtonClick()`: Navigates to the investment interface.
- `onCryptoButtonClick()`: Navigates to the cryptocurrency interface.
- `onVirementButtonClick()`: Navigates to the money transfer interface.
- `onLoanButtonClick()`: Navigates to the loan interface.
- `onLogOutButtonClick()`: Logs out the user and navigates to the login interface.
- `onMyAccountButtonClick()`: Generates and displays charts related to the user's account.
- `updateInfoTextArea()`: Updates the information displayed in the text areas based on the `UserObj` data.

3.3.3 Usage

The `UsersInterfaceController` class manages the main graphical user interface, ensuring the accurate display of user information and providing navigation to different functionalities. It integrates with other parts of the application, such as the `ChartGenerator` for chart visualization.

3.4 InvestmentController Class

The `InvestmentController` class, part of the `com.example.partjava` package, serves as the controller for the Investment.fxml interface. This controller manages investment-related functionalities, allowing users to view charts, invest in financial products, and navigate back to the main interface.

3.4.1 FXML Elements

- `returnButton:` Button for returning to the main user interface.

3.4.2 Methods

- `onViewGlobalButtonClick():` Displays global investment charts for the nearest 30 days.
- `onViewSelfButtonClick():` Displays user-specific investment charts for the nearest 30 days.
- `onInvestButtonClick():` Opens a new window to perform financial product investments.
- `onReturnButtonClick():` Returns to the main user interface.

3.4.3 Usage

The `InvestmentController` class facilitates interactions related to investments, including viewing investment charts and initiating financial product transactions. It interacts with the `ChartGenerator` class for chart visualization and utilizes the `SceneNavigator` class for seamless navigation between interfaces.

InvestController Class

The `InvestController` class, part of the `com.example.partjava` package, serves as the controller for the Invest.fxml interface. This controller handles the buying and selling of financial products, providing functionalities to search for stock prices, make purchases, and sell stocks.

FXML Elements

- `currencyLabel:` Label for displaying the user's available currency for investments.
- `companyNameField:` TextField for entering the name of the company for stock-related actions.
- `amountField:` TextField for entering the amount of stocks or currency for transactions.
- `passwordField:` PasswordField for entering the user's password to authorize transactions.

Methods

- `initialize():` Initializes the controller, updating the currency label.
- `onSearchButtonClick():` Searches for and displays the current stock price of a specified company.
- `onPurchaseButtonClick():` Processes the purchase of stocks based on user input.
- `onSellButtonClick():` Processes the sale of stocks based on user input.
- `onClearButtonClick():` Clears all input fields.
- `updateCurrencyLabel():` Updates the currency label with the user's available currency.
- `updateBankAcc(String accInfo):` Updates the user's bank account information using the provided string.

Usage

The `InvestController` class is responsible for managing user interactions related to financial product transactions. It validates user inputs, performs transactions, and updates relevant information, interacting with the `JavaClient`, `UserObj`, `GlobalObj`, and `SceneNavigator` classes as needed.

3.5 CryptoController Class

The `CryptoController` class, part of the `com.example.partjava` package, serves as the controller for the `Crypto.fxml` interface. This controller handles functionalities related to cryptocurrencies, including displaying sale and purchase lists, managing transactions, and refreshing data.

3.5.1 FXML Elements

- `refreshButton`: Button to refresh cryptocurrency data.
- `currencyLabel`: Label for displaying the user's available currency.
- `coinValueLabel`: Label for displaying the value of a cryptocurrency coin.
- `coinAmountLabel`: Label for displaying the amount of cryptocurrencies held by the user.
- `outputArea`: TextArea for displaying sale and purchase lists.
- `returnButton`: Button for returning to the main user interface.

3.5.2 Methods

- `showSaleList()`: Displays the list of sale orders.
- `showPurchaseList()`: Displays the list of purchase orders.
- `action()`: Opens a new window to interact with existing cryptocurrency orders.
- `order()`: Opens a new window to place a new cryptocurrency order.
- `refresh()`: Refreshes cryptocurrency data.
- `onReturnButtonClick()`: Returns to the main user interface.
- `initialize()`: Initializes the controller and retrieves cryptocurrency data.

CryptoActController Class

The `CryptoActController` class, part of the `com.example.partjava` package, serves as the controller for the `CryptoAct.fxml` interface. This controller manages actions related to existing cryptocurrency orders, including selling and buying actions.

FXML Elements

- `clientTextField`: TextField for entering the username of the client.
- `passwordField`: PasswordField for entering the user's password.

Methods

- `sellAction()`: Processes the sale action based on user input.
- `buyAction()`: Processes the buy action based on user input.

CryptoOrderController Class

The `CryptoOrderController` class, part of the `com.example.partjava` package, serves as the controller for the `CryptoOrder.fxml` interface. This controller handles placing sale and purchase orders for cryptocurrencies.

FXML Elements

- `amountTextField`: TextField for entering the amount of cryptocurrency for the order.
- `passwordField`: PasswordField for entering the user's password.

Methods

- `sellOrder():` Places a sale order or withdraws an existing sale order based on user input.
- `buyOrder():` Places a purchase order or withdraws an existing purchase order based on user input.

3.6 VirementController Class

The `VirementController` class, part of the `com.example.partjava` package, serves as the controller for the `Virement.fxml` interface. This controller handles functionalities related to fund transfers, allowing users to specify a beneficiary, enter an amount, and complete a transfer.

3.6.1 FXML Elements

- `currencyLabel:` Label for displaying the user's available currency.
- `beneficiaryField:` TextField for entering the username of the beneficiary.
- `amountField:` TextField for entering the amount to be transferred.
- `passwordField:` PasswordField for entering the user's password to authorize the transfer.
- `returnButton:` Button for returning to the main user interface.

3.6.2 Methods

- `initialize():` Initializes the controller, setting the currency label with the account's currency from `UserObj`.
- `transfer():` Processes the fund transfer based on user input, including beneficiary, amount, and password.
- `onReturnButtonClick():` Returns to the main user interface.

3.6.3 Usage

The `VirementController` class facilitates fund transfer interactions, validating user inputs, checking for errors, and updating `UserObj.account.currency` upon successful transfers. It interacts with the `JavaClient`, `Password2Hash`, `ShowAlert`, and `SceneNavigator` classes as needed.

3.7 LoanController Class

The `LoanController` class, part of the `com.example.partjava` package, serves as the controller for the `Loan.fxml` interface. This controller manages functionalities related to loan applications, allowing users to apply for loans, specifying the loan amount and entering the required password for authorization.

3.7.1 FXML Elements

- `currencyLabel:` Label for displaying the user's available currency.
- `debtLabel:` Label for displaying the user's current debt.
- `loanAmountField:` TextField for entering the loan amount.
- `passwordField:` PasswordField for entering the user's password to authorize the loan application.
- `returnButton:` Button for returning to the main user interface.

3.7.2 Constants

- `rate:` The interest rate for the loan application (0.0025 or 0.25%).
- `loanLimit:` The maximum allowable debt limit for a loan (-3000).

3.7.3 Methods

- `initialize():` Initializes the controller, setting labels with the current values from `UserObj`.
- `applyLoan():` Processes the loan application based on user input, including the loan amount and password.
- `onReturnButtonClick():` Returns to the main user interface.

3.7.4 Usage

The `LoanController` class facilitates loan application interactions, validating user inputs, checking for errors, and updating `UserObj.account.currency` and `UserObj.account.debt` upon successful loan applications. It interacts with the `JavaClient`, `Password2Hash`, `ShowAlert`, and `SceneNavigator` classes as needed.