# hLabs at Hesiod Financial, LLC

# Project Luca Documentation

Version 1.0

# Table of Contents

# Project Management

## Introduction

To anyone (at Hesiod or otherwise), welcome! This is the documentation for Project Luca, our Accounting tool, commissioned by the hLabs department of the Hesiod Financial group. The goal of this tool is to create a centralized application to give all members of Hesiod Financial the ability to easily access information on the finances and portfolio distributions as well as to give specialized members of Hesiod the ability to deposit and withdraw funds for investments and miscellaneous expenses.

The project began out of a desire for all members of the Hesiod Financial Investment Group to be able to move away from Excel/Google Sheets to something more sophisticated. This project also began as a general testing grounds for the members of the hLabs team to become acclimated to working in a large group setting, using version control and Agile SDLC methodologies, etc.

The backend was primarily developed in Java and Java plugins. The frontend was developed in Adobe XD to create the modern-looking user interface.

All of this has amounted towards a final product that not only employs innovative technologies and approaches to tracking operations in Hesiod, but towards a product that we are very proud of.

## Roles and Responsibilities

- *Jay Jayewardene* - Product Manager
- *Connor McMurry* - Backend Development Engineer
- *Ryan Tatton* - Backend Development Engineer
- *Thomas Patton* - Backend Development Engineer
- *Daniel Soares* - Frontend Development Engineer
- *Shaan Patel* -Quality Assurance Engineer

# Requirements Specifications

In order to utilize Luca, your computer must have Java installed. If you are unsure on how to do this, a good explanation can be found at Oracle's Java website: https://www.java.com/en/download/help/download_options.xml

Once this is done, you can find the latest version of Luca at hesiodfinancial.com/Luca

# Project Design and Implementation

Luca is divided into nine classes, all which are coded in Java. Below is a detailed description of each class, its connections to the other classes, as well as its constructor(s), nested classes, and methods/functions.

## Account Class

The `Account` class represents an account within the Luca application. Contained within an `Account` are the following: an associated `Bank` where funds currently not invested are stored, a `Portfolio` containing any `Asset(s)` being used in investing, and at least one `User` that can facilitate transactions between the `Bank` and `User`, and the `Bank` and `Portfolio`.

### Constructors

```
public Account(String name, Balance initialAmount, Bank b,
ArrayList<User> users, Portfolio port)
```

This will create an Account with an existing `Balance, Bank, Portfolio,` and `User(s)`

```
public Account(String accName, String bankName, String
portfolioName)
```

This will create an `Account` with an empty `Balance` and empty `Portfolio`, and uses Strings for naming conventions

```
public Account(String accName)
```

This will create an `Account` with only a name, and an `empty Balance, Bank, and Portfolio,` and no `User`. This option is not recommended unless it is not important to have names for the `Bank` and `Portfolio`.

### Methods

```
public double getCurrentAccountValue()
```

Calculates the net value in US Dollars of the Account and returns this in the form of a sum between the `Bank` US Dollar value and the `Portfolio` US Dollar value.

#### Return

The sum of the `Bank` US Dollar value and the `Portfolio` US Dollar value.

### Getter and Setter Methods

The getter methods are all static and have the following return values: `Balance, Bank, Portfolio,` Transaction Requests in the form of a LinkedList of `Transactions`, Transaction History in the form of a Stack of `Transactions`, Users in the form of an ArrayList of `User(s)`, name of `Account` and time the `Account` was created.

The setter methods are all static and void and change the following values: `Balance, Bank, Portfolio, User(s)` in the form of an ArrayList of `User(s)`, name of `Account` and time the `Account` was created. It is critical to note that we strongly advise against using any of these setter methods unless the `Account` does not already have the Object in question. In the case of the setter method for `Balance`, setting this will erase all previous transactions since these are directly tied to the `Balance` Object.

## Asset Class

The `Asset` class represents a stock or option that is owned by a `Portfolio`.

### Constructors

```
public Asset(String nameOfAsset, String sym, String sect, int
vol, double orgPrice, String orderType)
```

This will create an `Asset` to be added to its corresponding `Portfolio`.

#### Parameters

`nameOfAsset` - Full name of the `Asset`
`sym` – Asset symbol in accordance with New York Stock Exchange
`sect` – Sector to which the `Asset` belongs
`vol` – Number of shares of the `Asset`
`orgPrice` – Price at which the `Asset` is first acquired
`orderType` - Determines at what price and when the `Asset` is acquired or liquidated

### Methods

```
public void sellAsset()
```

Completes the sale of an `Asset` by setting the date and time of the sale, changing the `own` variable to false to signify that Hesiod no longer owns the `Asset`, calculates the returns on the sale of the `Asset` using the `setReturns()` and `calculateReturns()` methods and by calculating the time that the `Asset` was held by Hesiod. Finally, the `getPortfolio()` and

`remove()` methods are called on the `Portfolio` that initially held the `Asset` to formally remove it.

`public void coverAsset()`

Completes the buy to cover for closing out an existing short position on an `Asset` by setting the date and time of the sale, changing the `own` variable to false to signify that Hesiod no longer owns the `Asset`, calculates the returns on the covering of the `Asset` using the `setReturns()` and `calculateReturns()` methods and by calculating the time that the `Asset` was held by Hesiod. Finally, the `getPortfolio()` and `remove()` methods are called on the `Portfolio` that initially held the `Asset` to formally remove it.

`private double calculateReturns()`

Calculates the financial returns on an `Asset` by subtracting the `startPrice` (i.e. the acquisition price) from the `endPrice` (i.e. the liquidation price).

### Return

US Dollar amount; positive if gain or negative if loss on `Asset.`

`public double calculateHypotheticalReturns(double startPrice, double endPrice)`

Calculates the hypothetical financial returns on an `Asset` by subtracting the `startPrice` (i.e. the acquisition price) from the `endPrice` (i.e. the liquidation price). These prices are set by the parameters and are used by a method call to the `calculateReturns()` method.

### Parameters

`startPrice` - Price of the `Asset` at the time of acquisition/purchase
`endPrice` – Price of the `Asset` at the time of liquidation/sale

### Return

Hypothetical US Dollar amount; positive if gain or negative if loss on `Asset.`

`private double calculateTimeHeld()`

Calculates the amount of time between when the `Asset` is acquired and liquidated. This method makes use of the `ChronoUnit` Class from the Java API to find the difference between an acquisition and liquidation date.

#### Return

The amount of time that an `Asset` is held in seconds.

### Getter and Setter Methods

The getter methods have the following return values: `assetName, symbol, sector, volume, startPrice, endPrice, returns, startDate, endDate, timeHeld, own, orderType, acquistionTransaction` and `liquidationTransaction.` Note that the `isOwned()` function as a getter method for `own`.

The setter methods are all void and change the following values: `assetName, symbol, sector, volume, startPrice, endPrice, returns, startDate, endDate, timeHeld, own, orderType, acquistionTransaction` and `liquidationTransaction`.

## Balance Class

The `Balance` Class contains information about current and past values of an entity. `User(s), Portfolio(s),` and the `Bank` all have separate balances that are updated when transactions are requested and fulfilled. A `Stack` is used to maintain all balance statements, with the most recent one being at the top.

### Constructors

`public Balance(Transaction transaction)`

Creates a time-specific balance statement with an associated transaction.

#### Parameters

`transaction` - `Transaction` associated with the new balance statement

`public Balance(double amountToChange)`

Creates a time-specific balance statement of an entity with only an amount.

#### Parameters

`amountToChange` - Amount input into the balance (can be positive or negative)

`public Balance()`

Creates a zero-balance statement.

# Methods

`public void updateBalance(Balance newBalance)`

Uses stored balance statement values from `getBalanceHistory()` and adds `newBalance` to the current balance value to update the balance statement value.

### Parameters

`newBalance` - `Balance` used to update the current balance value

`public static Balance transferTo(Transaction transaction)`

Increases the `Balance` value by the amount determined by the `transaction` parameter. The new balance statement amount is then set to the transaction amount and returned to be added to the `balanceHistory.`

### Parameters

`transaction` – Transaction associated with updating the balance value.

### Return

Balance object to be added to the `balanceHistory`.

`public static Balance transferFrom(Transaction transaction)`

Decreases the `Balance` value by the amount determined by the `transaction` parameter. The new balance statement amount is then set to the transaction amount and returned to be subtracted to the `balanceHistory.`

### Parameters

`transaction` – Transaction associated with updating the balance value.

### Return

Balance object to be added to the `balanceHistory`.

## Getter and Setter Methods

The getter methods have the following return values: current or any `balanceAmount, balanceTimeStamp, associatedTransaction, balanceHistory.`

The setter methods are all void and change the following values: any `balanceAmount, balanceTimeStamp, associatedTransaction, balanceHistory.` Changing the `balanceHistory` is only suggested if the `balanceHistory` has not be instantiated, since doing so will erase any previous data in the `Stack<Balance>.`

# Bank Class

The `Bank` Class represents the bank where all inactive trading funds are kept. Unless it is manually assigned a name by the user, it is default called "Unnamed Bank." The bank has a `Balance` that contains both the current value of the bank, as well as past values. Each time the bank's balance is updated, the time in which it is updated, the amount, and the associated transaction are all recorded as part of the new balance. Users can deposit funds from external sources; this will add value to the bank but does not affect the `Portfolio` balance. Users can also withdraw funds and have them transferred to whomever requested the withdrawal.

## Constructors

### `Bank()`

Creates a bank with the default name, "Unnamed Bank," and has a balance of 0.

### `Bank(String name)`

Creates a bank with only a non-default name, but no initial amount.

#### Parameters

`name` - User chosen name for `Bank`

### `Bank(String name, Balance balance)`

Creates a bank with a non-default name and an existing Balance.

#### Parameters

`name` - User chosen name for `Bank`
`balance` - Contains current and past values, transactions and timestamps

### `Bank(String name, double initialAmount)`

Creates a bank with a non-default name and some initial value, but no associated transaction.

#### Parameters

`name` - User chosen name for `Bank`
`balance` - Dollar amount in bank account, but not a deposit

### `Bank(String name, Transaction firstTransaction)`

Creates a bank with a non-default name and a Transaction.

#### Parameters

`name` - User chosen name for `Bank`

`firstTransaction` - Transaction that sets the bank

## Methods

```
public static void deposit(Transaction transaction)
```

Deposits money into the bank account.

#### Parameters

`transaction` - Associated with the deposit

```
public static void withdraw(Transaction transaction)
```

Withdraws money from the bank and is transferred to the request user's balance, if not denied.

#### Parameters

`transaction` - Associated with the withdrawal

### Getter and Setter Methods

The getter methods have the following return values: `bankBalance` and `bankName.`

The setter methods are all void and change the following values: `bankBalance` and `bankName.`

# Encryption Class

This class uses methods used throughout the LASER program.

## Methods

```
public static String applySHA256(String trx)
```

This method utilizes a 256 bit secure hashing algorithm for encryption.

```
public static byte[] applySignature(PrivateKey privateKey, String input)
```

Signs the transaction with the private key to identify the address.

```
public static boolean verifySignature(PublicKey publicKey, String data, byte[] signature)
```

Verifies a transaction came from a specific public key.

`private static SecretKeySpec sks`

`private static byte[] key`

`public static void setKey(String myKey)`

Inputs a string as the information to be entered.

`public static String encrypt(String strToEncrypt, String secret)`

This encrypts the information using the private key.

`public static String decrypt(String strToDecrypt, String secret)`

Decrypts the hashes.

`public static KeyPair generateKeyPair()`

The key pairs generate by this method are used to sign transactions made by an address.

# LucaLogin Class

This class authenticates the `User` via a username and password.

## Methods

`public static void main(String[] args)`

Attempt to authenticate the `User`.

### Parameters

`args` - Input argument for this application; these are ignored.

## Nested Classes

`class MyCallbackHandler implements CallbackHandler`

Luca uses this class to implement the CallbackHandler. This is a text-based application. Therefore, it displays information to the user using the OutputStreams System.out and System.err and gathers input from the user using the InputStream System.in.

### Methods

`public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException`

Invokes an array of Callbacks

### Parameters

`callbacks` - An array of `Callback` objects which contain the information requested by an underlying security service to be retrieved or displayed.

### Throws

`IOException` - If an input or output error occurs
`UnsupportedCallbackException` - If the implementation of this method does not support one or more of the Callbacks specified in the `callbacks` parameter.

# Portfolio Class

Contains all assets owned by the `Account`. `LinkedHashSet` is used here so as to preserve the order in which the assets were acquired. This is useful in the case where two acquisition transactions of the same asset are resolved at different times with different prices, so each can be handled uniquely.

## Constructors

```
public Portfolio(String name, Balance portfolioBalance,
LinkedList<Asset> portfolio)
```

Creates a portfolio with a non-default name, balance and set.

### Parameters

`name` - Designated name for the portfolio
`portfolioBalance` - Contains current and previous dollar amounts inputted into the portfolio when assets are initially acquired.
`portfolio` - Contains all owned assets in the order in which they were acquired

```
public Portfolio(String name, Balance portfolioBalance)
```

Creates a portfolio with a non-default name, a potentially zero balance, and an empty portfolio with no owned assets.

### Parameters

`name` - Designated name for the portfolio
`portfolioBalance` - Contains current and previous values, associated transactions, and timestamps.

```
public Portfolio(String name)
```

Creates a portfolio with a non-default name, a balance of zero, and an empty portfolio with no owned assets.

### Parameters

`name` - Designated name for the portfolio

`public Portfolio()`

Creates a portfolio with the default name of "Unnamed Portfolio", a balance of zero, and an empty portfolio with no owned assets.

## Methods

`public static void buyOrder(Transaction transaction)`

Acquires the asset associated with a "BUY" transaction request. If the request is not denied during resolution, the start date and owned statuses of the asset belonging to the transaction are set. The status of the transaction is set to "BOUGHT". The amount associated with buying the asset is transferred from the `Bank` to the portfolio. If the amount required to buy the asset is greater than that which resides in the `Bank`, a message is printed, and the transaction status is set to "CANCELLED".

### Parameters

`transaction` - Contains the asset to be bought

`public static void sellOrder(Transaction transaction)`

Liquidates a bought asset if the asset currently exists in the portfolio.

### Parameters

`transaction` - Contains the asset to be sold

`public static void shortOrder(Transaction transaction)`

Shorts an asset associated with the transaction on the condition that there are enough funds in the `Bank` to immediately cover all shorted shares of the asset. This implemented for risk-prevention purposes. If the amount associated with the transaction is less than what is in the bank, then the short order is processed such that the resolution date and time of the transaction is set, the asset start date is set to when the short order is processed, the transaction status is set to "SHORTED," the asset is set to owned, and the portfolio balance is updated. The bank balance is not updated since no returns have yet been gained or lost.

#### Parameters

`transaction` - Contains the asset to be shorted

`public static void coverOrder(Transaction transaction)`

Covers a shorted asset based on the condition that the asset is currently being shorted in the portfolio. The resolution date is set to when the asset is covered. The transaction status is set to "COVERED." Asset returns and duration of owning the asset are calculated. The asset is set to not owned and is removed from the portfolio. The portfolio and bank balances are updated accordingly. Because of the nature of shorting and covering stocks, the amount set to update the bank balance is opposite that which was returned.

#### Parameters

`transaction` - Contains the asset to be covered

`public static void addToPortfolio(Asset asset)`

Adds an asset to the portfolio of owned assets. Used during the `buyOrder(Transaction)` and `shortOrder(Transaction)` methods.

#### Parameters

`asset` - Asset to be added to the portfolio

### Getter and Setter Methods

The getter methods have the following return values: `portfolio, nameOfPortfolio, portfolioBalance, timeCreated.`

The setter methods are all void and change the following values: `portfolio, nameOfPortfolio, portfolioBalance, timeCreated.` We strongly advise against using any of these methods to change their corresponding variables unless you are presented with one of the following cases: you have not set the `portfolio`, you have not set the `nameOfPortfolio`, you have not set the `portfolioBalance`, or the `timeCreated` is incorrect.

## Transaction Class

Essentially, this class will demonstrate a movement of funds of funds between any two account entities, whether that is between the user and the bank, or the bank and the portfolio. Every `User` must first request a transaction, before the funds are reallocated. An admin `User` is able to resolve the transaction such that the funds or asset associated with the transaction is either approved or denied. Also, an admin `User` who requests a transaction is able to resolve the transaction immediately. In either case, to request or to resolve, the `User` is asked to confirm the decision to help minimize accidental `User` action, particularly when resolving a

transaction. When a transaction is first requested, the request `User`, the amount, the date and time requested, the type of transaction, the status of the transaction, and the transaction ID are set. By default, the status of a requested transaction is set to OPEN. If the `User` requesting the transaction is a non-admin, the request is added to the list of requests `Account` to be resolved. When a transaction is resolved by an admin `User` the `resolveUser`, `resolveDate`, and the `transactionStatus` are updated. If the status is set to DENIED, the request is removed from the list of transaction requests (if requested by a non-admin `User`) and added to the transaction history `Account`) without any funds being reallocated. If transaction is neither DENIED nor CANCELLED (set when either the request or resolution is being made, the funds or `Asset` associated with transaction are processed, and the appropriate `Balance` are updated.

## Constructors

### `public Transaction(double amount, String type)`

Creates a transaction with no associated `Asset`.

#### Parameters

`amount` - Dollar amount requested; Always positive
`type` - Determines if the transaction is related to the `Bank` and/or `Portfolio`.

### `public Transaction(String type, Asset transactionAsset)`

Creates a transaction with an associated `Asset`.

#### Parameters

`type` - Since there is an associated `Asset`, this will be BUY, SELL, SHORT, or COVER
`amount` - `Asset` associated with the transaction

## Methods

### `public long generateID()`

Creates a numerical identifier for a transaction based on the date and time in which the transaction is requested.

#### Return

Numerical date-time identifier based on when the transaction is requested.

### `public boolean confirmAction()`

Requires that the `User` confirms the action to either request or resolve a transaction.

### Return

True if the response if "yes," and false if "no."

`public enum Status`

The default is OPEN. This is updated when the admin `User` resolves the transaction.

`public enum Type`

This is set by the user when requesting the transaction. DEPOSIT and WITHDRAW are reserved for `Bank` only transactions, while BUY, SELL, SHORT and COVER are reserved for `Portfolio` only transactions.

`public Status getMatchingStatus()`

Used when an admin {@link User} is requesting a transaction. Since an admin `User` can request and resolve transactions immediately, to is more efficient and less prone to `User` error if the status of the transaction is updated to match the type of transaction as opposed to entering the updated status when resolving it.

### Return

Status corresponding to the type of the transaction. Useful for automatically updating the status of a transaction without having it as a method parameter.

`public void addTransactionRequest()`

Adds a new transaction request to the list of requested transactions.

`public void removeTransactionRequest()`

Deletes a requested transaction from the list of requested transactions. This should only be used when a transaction has been requested and is either resolved or cancelled.

`public void addToTransactionHistory()`

Adds a resolved transaction to the history of previously resolved transactions.

## Getter and Setter Methods

The getter methods have the following return values: `requestUser, resolveUser, transactionAmount, requestDate, resolveData, transactionStatus, transactionType, transactionID, transactionAsset, userPublicKey, userPrivateKey, signature, timestamp, transactionData.`

The setter methods are all void and change the following values:
`requestUser, resolveUser, transactionAmount, requestDate, resolveData, transactionStatus, transactionType, transactionID, transactionAsset, userPublicKey, userPrivateKey, signature, timestamp, transactionData.`

# User Class

User associated with the `Account`. Each user has a *username*, which is based on the user's first, middle, and last initials. The user's *password* is randomly generated via `java.security`. A user with admin permissions has the ability to resolve requested transactions, change the status of other users, and make changes to the `Account`, such as adding `addUser(User)` or removing `removeUser(User)` users. Each user has a `userBalance`, which can be used to keep track of outstanding dues or annual dividends. Each user also has a `userContribution`, which comprises the `Account` funds. The date and time in which the user is created is assigned to `timeCreated`.

## Constructors

```
public User(String first, String middleInit, String last, String password, String w1, String w2, String w3, UserType role, double contribution)
```

## Methods

```
public String makeUsername(String first, String middle, String last)
```

Creates a username for the user that is the initials of first, middle, and last names, and three randomly generated numbers.

### Parameters

`first` - Initial of the first name of the user
`middle` - Initial of the middle name of the user
`last` - Initial of the last name of the user

### Return

Username.

```
public double calculatePctHoldings()
```

Calculates the percent of total contributions that the user in question has contributed.

### Return

The percent that the user holds of the total portfolio.

`public double roundToThree(double num)`

Rounds the number in question to three decimal places.

### Parameters

`num` - Number to be rounded to three decimal places

### Return

The rounded number.

`public double calculateHoldingsValue()`

Calculates the amount of cash in an `Account` that a user has.

### Return

The calculated amount of cash held by a user.

`public void requestTransaction(Transaction request)`

Request a `Transaction`. All users can request transactions, which are stored in a Linked List in the `Transaction` class. If a user is an admin, the request is immediately resolved. Otherwise, the request is added to the transaction request list.

### Parameters

`request` - Requested transaction

`public void resolveTransaction(Transaction transaction, String updatedStatus)`

Resolves a requested `Transaction`. Only an admin `User` can resolve requests.

### Parameters

`transaction` - Requested `Transaction` to be resolved
`updatedStatus` - Status after the `Transaction` is resolved

`public void contribute(double amount)`

Allows a user to contribute more towards investing.

### Parameters

`amount` - Dollar amount to add to the `Account`

`public void addUser(User newUser)`

Adds a user to the `Account`. Only available to admin users.

#### Parameters

`newUser` - User to be added

`public void removeUser(User userToRemove)`

Removes a user to the `Account`. Only available to admin users.

#### Parameters

`userToRemove` - User to be removed

`public void changeClearance(User otherUser, UserType newType)`

Allows a user's status to be changed to either regular or admin.

#### Parameters

`otherUser` - The user whose clearance is to be changed
`newType` - The new clearance level to be assigned to the user

`public enum UserType`

Defines the different types of clearance levels based on an integer ranking between 4 and 1, with 4 having the highest clearance level, and 1 having the lowest.

### Getter and Setter Methods

The getter methods have the following return values: `username, password, firstInit, middleInit, lastInit, timeCreated, userPrivateKey, userPublicKey, clearance, userContribution, userBalance.`

The setter methods are all void and change the following values: `username, password, firstInit, middleInit, lastInit, timeCreated, userPrivateKey, userPublicKey, clearance, userContribution, userBalance.` We strongly advise against changing `username, timeCreated,` and `userContribution.`

# Acknowledgements

We at hLabs would like to thank everyone at Hesiod Financial for their continued support throughout the arduous yet rewarding and incredible experience of creating Luca.