Documentation

Approach:

When developing, I used the following process: (Associated requirements list and design is found below)

1. Analyze assessment requirements. Break them down into a succinct checklist.
2. Create pseudocode design of major methods.
3. Scaffold methods & add Javadoc comments.
4. Implement design.
5. Create final documentation of build process, upload to github repository, & submit.

During step 4 (implementation), I took the following process:

1. Get build working for all classes.
2. Wire up all functions to get basic structure (i.e. X calls Y. Y calls Z). Verify that the full call stack is being executed.
3. Complete a single function (working from highest level functions to lowest level).
4. Test that the previous function works as desired.
5. If the system is not complete, return to step 3.
6. Full requirements testing.
7. Cleanup, refactoring, and error messaging. Update names to be more relevant.
8. Repeat full requirements testing.

Libraries used:

- *Jsoup*: I used this package to manage the HTML file. I chose Jsoup for its fluent syntax, high readability, and popularity.
- *java.io.**: I used buffered readers & writers to read from the console and write to the account log file. I chose to use them because of their common use for file IO and command line IO.
- *java.util.ArrayList*: I chose to use ArrayList throughout (over arrays or other options) for the increased readability and built in functionality. Using built in functions such as '.contains()' and '.add()' not only make the code easier to understand in-line, but also remove the need to reproduce the functionality with lower-level data structures.

How to Build/Run:

1. Install JDK version 14.0.1 (can be found here https://www.oracle.com/java/technologies/javase-jdk14-downloads.html)
2. Download all files from github repository
3. Open Windows command prompt and run the following commands
    a. javac -cp ".;./bin/jsoup-1.13.1.jar" BankAccountSystem.java
    b. java -cp ".;./bin/jsoup-1.13.1.jar" BankAccountSystem

Requirements:

[X] – Is a command-line program

[X] – Can make deposits

[X] – Can make withdrawals

[X] – Can check current account balance

[X] – Upon startup, print text "Please enter in a command (Deposit, Withdraw, Balance, Exit) :" & wait for input

If user inputs "Deposit":

[X] – Print text "Please enter an amount to deposit:" & wait for input

[X] – Rejects negative input & prompts user for input again

[X] – Rejects input with >2 decimal places & prompts user for input again

[X] – Upon valid input, append given number to end of the log.html file

If user inputs "Withdraw":

[X] – Print text "Please enter an amount to withdraw:" & wait for input

[X] – Rejects negative input & prompts user for input again

[X] – Rejects input with >2 decimal places & prompts user for input again

[X] – Upon valid input, change given number to negative and append it to end of the log.html file

If user inputs "Balance":

[X] – Read contents of log.html file & determine the sum of the values.

[X] – Prints the current balance in the form: "The current balance is: ${balance}"

[X] – After process for "Deposit", "Withdraw" and "Balance", print the startup text ("Please enter in a command (Deposit, Withdraw, Balance, Exit) :")

[X] – At any time, if the use enters "Exit", the program terminates

[X] – All input is case-insensitive

[X] – At any time, if the user enters an invalid input, they should be prompted to input again

[X] – All methods have Javadoc comments that document (1) the purpose of each parameter and (2) all possible outcomes

[X] – Somewhere, there is an explanation of (1) approach and (2) why the libraries used are used.

[X] – Everything required to build and run code is in a github.com repository

[X] – Includes instructions on building and running program from the command line

Initial Design:

void BankAccountSystem.Main()

```
{
    Startup();
}
```

Enum Values: Start, Deposit, Withdraw

void BankAccountSystem.Startup()

```
{
    While (true)
        userInput = PromptUserForInput(Enum.Start)
        switch(userInput)
            Deposit:
                Deposit()
            Withdraw:
                Withdraw()
            Balance:
                GetBalance()
}
```

string UserInputService.PromptUserForInput(Enum)

```
{
    while (waitingForValidInput)
        PrintInputPrompt(Enum)
        input = Read line from buffer reader
        If (input == exit)
            Exit()
        if (IsValidInput(Enum, input))
            return input
        Print invalid input message
}
```

void UserInputService.PrintInputPrompt(Enum)

```
{
    switch(Enum)
        Start:
            Print start message "Please enter in a command (Deposit, Withdraw, Balance, Exit) :"
        Deposit:
            Print deposit message "Please enter an amount to deposit:"
        Withdraw:
            Print withdraw message "Please enter an amount to withdraw:"
}
```

bool UserInputService.IsValidInput(Enum, string input)

```
{
    isValid = false
```

```
    if Enum.Start
        isValid = validInputValues.Contains(input.ToLowerCase())
    else
        isValid = (input is <= 2 decimals) AND (input is non-negative)
    return isValid
}

void UserInputService.Exit()
{
    Print "Program shutting down. Thank you"
    System.Exit
}

void Account.Deposit()
{
    depositAmount = PromptUserForInput(Enum.Deposit)
    Convert deposit amount to float
    AppendToFile(depositAmount)
}

void Account.Withdraw()
{
    withdrawAmount = PromptUserForInput(Enum.Withdraw)
    Convert withdraw amount to float
    AppendToFile(withdrawAmount * -1)
}




void Account.GetBalance()
{
    balance = ReadBalanceFromFile()
    Print "The current balance is: ${balance}"
}

float Account.ReadBalanceFromFile()
{
    transactions = ReadAllTransactionsFromFile()
    return sum of transactions
}

collection of floats FileService.ReadAllTransactions()
{
    transactions = read from file, ignore non-number lines
    return transactions
}
```

```
void FileService.AppendToNewLine(string)
{
    Write string to end of file on a new line
}
```