Errors - WordCount

The only error I had in my process was in this block of code:

```
try {
    // Create a Scanner to read the file
    Scanner fileScanner = new Scanner(new File(filePath));

    // Use a regular expression to match words
    while (fileScanner.hasNext()) {
        // Read the next sequence of words from the file
        String word = fileScanner.next();

        // Match only words (letters a-z or A-Z)
        word = word.replaceAll("[^a-zA-Z]", "");

        if (!word.isEmpty()) {
            wordCount++;
            totalWordLength += word.length();
        }
    }
```

In this method, the application would only count the letters of all words that were on the same line, but would not account for words on separate lines.

I resolved this by adjusting the logic of the code to look like this:

```
try {
    // Create a Scanner to read the file
    Scanner fileScanner = new Scanner(new File(filePath));

    // Process the file, word by word
    while (fileScanner.hasNext()) {
        // Read the next sequence of characters (including hyphenated words)
        String word = fileScanner.next();

        // Split by hyphen to treat hyphenated words as separate words
        String[] splitWords = word.split("-");

        // Iterate over the split words
        for (String splitWord : splitWords) {
            // Remove any non-letter characters (punctuation, numbers, etc.)
            splitWord = splitWord.replaceAll("[^a-zA-Z]", "");

            // If the split word is a valid word (not empty), count it
            if (!splitWord.isEmpty()) {
                wordCount++;
                totalWordLength += splitWord.length();
            }
        }
    }
```

Now, the application goes through the list of words in the source text line by line and also accounts for cases such as multiple words on the same line as well as on separate lines.

Errors - FindAndReplace

```
    // Using StringBuilder to store file content for efficient replacement
    StringBuilder fileContent = new StringBuilder();
    Scanner fileScanner = new Scanner(file);

    // Read the file line by line
    while (fileScanner.hasNextLine()) {
        String line = fileScanner.nextLine();
        fileContent.append(line).append(System.lineSeparator()); // Preserve line breaks
    }
```

```
    fileScanner.close();

    // Perform the replacement
    String updatedContent = fileContent.toString().replace(searchWord, replacementWord);

    // Write the updated content back to the file
    FileWriter writer = new FileWriter(file);
    writer.write(updatedContent);
    writer.close();

    System.out.println("Replacement completed successfully.");
```

Originally, I did not use a `StringBuilder` to store the file content. Without the `StringBuilder`, reading and modifying the file line by line would require handling strings more directly, which can be less efficient.

By adding the `StringBuilder`, I was able to efficiently accumulate and modify the file's content without creating multiple intermediate string objects, making the replacement process smoother and faster.

I made no other errors in my process.

Errors - MySavings

One error I encountered during my process was while coding the PiggyBank class of my application.

```
// Remove coins
public void removePennies(int count) {
    pennies -= count;
}

public void removeNickels(int count) {
    nickels -= count;
}

public void removeDimes(int count) {
    dimes -= count;
}

public void removeQuarters(int count) {
    quarters -= count;
}

// Calculate total amount in dollars
public double getTotal() {
    return pennies * 0.01 + nickels * 0.05 + dimes * 0.10 + quarters * 0.25;
}
```

The error here is that if I enter an amount of coins that is more than is actually in the file, I will end up with a negative amount of money.

This idea could be applied to adding features related to loans to the application, however this is not what I wanted.

If the user tries to take out, for example, a nickel when they only have one penny, the application *should* read the number of nickels (which would be zero) and remove that amount, therefore making no actual change to the balance.

```
--- My Savings Menu ---
1. Add Pennies
2. Add Nickels
3. Add Dimes
4. Add Quarters
5. Remove Pennies
6. Remove Nickels
7. Remove Dimes
8. Remove Quarters
9. View Total Savings
0. Exit and Save
Enter your choice: 7
Enter the number of dimes to remove: 1

--- My Savings Menu ---
```

```
1. Add Pennies
2. Add Nickels
3. Add Dimes
4. Add Quarters
5. Remove Pennies
6. Remove Nickels
7. Remove Dimes
8. Remove Quarters
9. View Total Savings
0. Exit and Save
Enter your choice: 9
Total Savings: $-0.1
```

(Fixed)

```
7. Remove Dimes
8. Remove Quarters
9. View Total Savings
0. Exit and Save
Enter your choice: 9
Total Savings: $0.0

--- My Savings Menu ---
1. Add Pennies
2. Add Nickels
3. Add Dimes
4. Add Quarters
5. Remove Pennies
6. Remove Nickels
7. Remove Dimes
8. Remove Quarters
9. View Total Savings
0. Exit and Save
Enter your choice: 1
Enter the number of pennies: 1

--- My Savings Menu ---
1. Add Pennies
2. Add Nickels
3. Add Dimes
4. Add Quarters
5. Remove Pennies
6. Remove Nickels
7. Remove Dimes
8. Remove Quarters
9. View Total Savings
0. Exit and Save
Enter your choice: 6
Enter the number of nickels to remove: 1

--- My Savings Menu ---
1. Add Pennies
2. Add Nickels
3. Add Dimes
4. Add Quarters
5. Remove Pennies
6. Remove Nickels
7. Remove Dimes
8. Remove Quarters
9. View Total Savings
0. Exit and Save
Enter your choice: 9
Total Savings: $0.01
```

Another error I made in my process was relating to the file features (uploading, loading, etc.)

Initially, I tried to manually write and read each field (pennies, nickels, dimes, and quarters) separately using `PrintWriter` and `Scanner`.

```java
private static void savePiggyBank(PiggyBank bank) {
    try (PrintWriter writer = new PrintWriter(new FileWriter(FILE_NAME))) {
        writer.println(bank.getPennies());  // Write pennies to file
        writer.println(bank.getNickels());  // Write nickels to file
        writer.println(bank.getDimes());    // Write dimes to file
        writer.println(bank.getQuarters()); // Write quarters to file
        System.out.println("PiggyBank saved using basic file I/O.");
    } catch (IOException e) {
        System.out.println("Error saving PiggyBank: " + e.getMessage());
    }
}
```

(Example of code for functionality of saving the piggybank, the code for functionality of loading the piggy bank was similar)

To fix my code and make it more efficient, I switched to using `ObjectOutputStream` and `ObjectInputStream`.

This allowed me to serialize the entire PiggyBank object and save it to a file, making the process simpler.

```java
// Method to save the PiggyBank object to a file using output stream
private static void savePiggyBank(PiggyBank bank) {
    try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(FILE_NAME))) {
        oos.writeObject(bank);
        System.out.println("PiggyBank saved successfully.");
    } catch (IOException e) {
        System.out.println("Error saving PiggyBank: " + e.getMessage());
    }
}

// Method to load the PiggyBank object from a file using input stream
private static PiggyBank loadPiggyBank() {
    try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(FILE_NAME))) {
        return (PiggyBank) ois.readObject();
    } catch (IOException | ClassNotFoundException e) {
        System.out.println("No saved PiggyBank found. Starting a new one.");
        return new PiggyBank(); // If the file doesn't exist, return a new PiggyBank
    }
}
```

Using `ObjectOutputStream` and `ObjectInputStream`, I could easily save and load the entire object without managing each field individually. This removed the need for field-by-field I/O and ensured that the entire object state was preserved.