Reflection - WordCount

```
public class WordCount {
    public static void main(String[] args) {
        // File path to the source.txt file
        String filePath = "C:/Users/38020001/git/CS30P3F2024/chapter11/src/mastery/source.txt";

        int wordCount = 0;
        int totalWordLength = 0;
```

First, I declared all my variables:

`wordCount` - the number of words

`totalWordLength` - the total amount of alphabetical characters in all the words

`filePath` - the path to the source text file

```
    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 word (sequence of letters)
            String word = fileScanner.findInLine("[a-zA-Z]+");
            if (word != null) {
                wordCount++;
                totalWordLength += word.length();
            } else {
                // Skip non-letter characters
                fileScanner.next();
            }
        }

        // Close the scanner
```

Next, I added a scanner that reads the contents of the file located at `filePath`.

Then, I added `fileScanner.hasNext()`:This checks if there is more content (a "next" token) to read in the file. It returns `true` if there is another token, allowing the `while` loop to keep iterating until the entire file is processed.

Next, the method fileScanner.findInLine("[a-zA-Z]+") looks for the next sequence of characters that match the regular expression [a-zA-Z]+.
`[a-zA-Z]` matches any single letter, whether it's uppercase or lowercase. It excludes numbers, punctuation, spaces, and any other characters that are not letters.
The findInLine method only searches within the current line of the file. If no match is found on that line, it doesn't search further (which ended up causing an issue if the user expects the application to process words across lines).

I came to use [a-zA-Z]+ after doing research online, seeking to optimize my code.

To resolve the issue of the program only reading one line, I modified my 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 code accounts for many cases, such as:
- Hyphenated words (considered two words)
- Multiple words on the same line
- Words separated by line
- Some words separated by line, some on the same line
- etc.

```
            // Close the scanner
            fileScanner.close();

            // Display the results
            System.out.println("Number of words: " + wordCount);
            if (wordCount > 0) {
                double averageWordLength = (double) totalWordLength / wordCount;
                System.out.printf("Average word length: " + averageWordLength);
            } else {
                System.out.println("No words found.");
            }

        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + filePath);
        }
    }
```

The rest of the code handles the output to console and the exception of no source file found.

Reflection - FindAndReplace

First, I imported the `java.io.*` and `java.util.Scanner` libraries.
I then created a `Scanner` object to read input from the user.
The user is then prompted for the file path, where the file to be modified is located.

```
package mastery;

import java.io.*;
import java.util.Scanner;

public class FindAndReplace {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        // Prompt for the file name
        System.out.print("Enter the file name (with path): ");
        String fileName = input.nextLine();
```

Next, the user is asked to input the word or phrase they want to search for and what they want to replace it with.

```
// Prompt for the word/phrase to search for
System.out.print("Enter the word/phrase to search for: ");
String searchWord = input.nextLine();

// Prompt for the word/phrase to replace with
System.out.print("Enter the replacement word/phrase: ");
String replacementWord = input.nextLine();
```

After all of these variables are set by the user, the next part of the program handles reading the file, replacing the words, and writing the changes back to the file. I used a try-catch block to manage any potential `IOException` errors.

```
// Read the file content, perform replacement, and write back to the file
try {
    // Read the file content
    File file = new File(fileName);
    if (!file.exists()) {
        System.out.println("File not found: " + fileName);
        return;
    }
```

(Reading the file content; outputting an exception to the console if the file is not found.)

Next, I used a `StringBuilder` to efficiently accumulate the content of the file.

I found with research that this was the most efficient method to go about this process.

```
            // 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.");

        } catch (IOException e) {
            System.out.println("An error occurred while processing the file.");
            e.printStackTrace();
        }

        input.close();
    }
}
```

Once all the content is accumulated in the `StringBuilder`, I replaced the target word with the new word.

The updated content is then written to the file and a result message is output to the console.

Finally, in the `catch` block, I handled any possible `IOException` errors that could occur, such as issues with file access.

Reflection - MySavings

First, in the MySavingsFile class, I imported the `java.io.*` and `java.util.Scanner` libraries and declared the file path

Inside the main method, a `Scanner` object is created for user input.

The program attempts to load an existing `PiggyBank` object from the file by calling `loadPiggyBank()`. If the file doesn't exist, it creates a new `PiggyBank` object.

```
private static String FILE_NAME = "C:/Users/38020001/git/CS30P3F2024/chapter11/src/mastery/PiggyBank.txt";

public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    PiggyBank bank = loadPiggyBank(); // Load PiggyBank from file or create a new one
```

A boolean `exit` is initialized to false, and a `while` loop starts. This loop continues until the user chooses to exit the program.

```java
boolean exit = false;
while (!exit) {
    // Display the menu
    System.out.println("\n--- My Savings Menu ---");
    System.out.println("1. Add Pennies");
    System.out.println("2. Add Nickels");
    System.out.println("3. Add Dimes");
    System.out.println("4. Add Quarters");
    System.out.println("5. Remove Pennies");
    System.out.println("6. Remove Nickels");
    System.out.println("7. Remove Dimes");
    System.out.println("8. Remove Quarters");
    System.out.println("9. View Total Savings");
    System.out.println("0. Exit and Save");
    System.out.print("Enter your choice: ");
```

This while loop is what re-shows the prompt every time the user completes one of the functions of the application.

```java
    int choice = input.nextInt();

    switch (choice) {
        case 1:
            System.out.print("Enter the number of pennies: ");
            bank.addPennies(input.nextInt());
            break;
        case 2:
            System.out.print("Enter the number of nickels: ");
            bank.addNickels(input.nextInt());
            break;
        case 3:
            System.out.print("Enter the number of dimes: ");
            bank.addDimes(input.nextInt());
            break;
        case 4:
            System.out.print("Enter the number of quarters: ");
            bank.addQuarters(input.nextInt());
            break;
        case 5:
            System.out.print("Enter the number of pennies to remove: ");
            bank.removePennies(input.nextInt());
            break;
        case 6:
            System.out.print("Enter the number of nickels to remove: ");
            bank.removeNickels(input.nextInt());
            break;
        case 7:
            System.out.print("Enter the number of dimes to remove: ");
            bank.removeDimes(input.nextInt());
            break;
        case 8:
            System.out.print("Enter the number of quarters to remove: ");
            bank.removeQuarters(input.nextInt());
            break;
        case 9:
            System.out.println("Total Savings: $" + bank.getTotal());
            break;
        case 0:
            exit = true;
            savePiggyBank(bank); // Save the PiggyBank to a file
            break;
        default:
            System.out.println("Invalid choice! Please select again.");
    }
}
```

```
    input.close();
```

Next, the program takes an input from the user on their choice of what they want to run on the application and displays a corresponding prompt
Based on their input, a respective function is performed in the `PiggyBank` class.

```
public class PiggyBank implements Serializable {
    private int pennies;
    private int nickels;
    private int dimes;
    private int quarters;

    // Constructor
    public PiggyBank() {
        pennies = 0;
        nickels = 0;
        dimes = 0;
        quarters = 0;
    }

    // Add coins
    public void addPennies(int count) {
        pennies += count;
    }

    public void addNickels(int count) {
        nickels += count;
    }

    public void addDimes(int count) {
        dimes += count;
    }

    public void addQuarters(int count) {
        quarters += count;
    }

    // Remove coins
    public void removePennies(int count) {
        pennies -= Math.min(count, pennies); // Ensure not removing more than we have
    }

    public void removeNickels(int count) {
        nickels -= Math.min(count, nickels);
    }

    public void removeDimes(int count) {
        dimes -= Math.min(count, dimes);
    }

    public void removeQuarters(int count) {
        quarters -= Math.min(count, quarters);
    }

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

Lastly, when the user exits (or when the program is first opened), these methods are responsible for the file functions (saving and loading the bank info)

```
    // 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());
```

```
            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
        }
    }
```

More explanation for the usage of `OOS` and `OIS` can be found in my error log. (This was the only major change in functionality I made)