General description

This application offers the functionality of identifying and grouping anagrams together. Given an input text file, which contains words in the format of one word per line, the program processes each word and decides to which group it belongs. This is possible by using the *map* data structure.

Implementation  
The program is divided in 3 separate classes: Main.java, WordReader.java and AnagramFinder.java. This is made in order to assure **maintainability** and to provide a clean code. By separation of logic in different classes, the Single Responsibility principle is respected. It keeps the code clear and modular, leading to the possibility of extending and upgrading some features without interacting with the others.

The Main.java class is responsible with the initialization of components and the printing of output.

The WordReader.java class is responsible with the reading of the input. It contains *readWordsFromFile* method, that allows reading from a file. The method uses BufferedReader, which is perfect for this program, allowing reading line by line. There is a try-with-resources structure in this method, in order to handle possible errors.

The AnagramFinder.java class performs the main logic of the program, by mapping any word to its anagram. This is possible because of sort of letters. Every read word is processed so that its letters are sorted, creating the “primary” anagram. If this is new, it is put as a new key in the HashMap structure, otherwise, the word is added to the already existing group of same key. Using a HashMap structure allows a fast access to any group, with a O(1) complexity for lookup. This is one of the key points in making the program **scalable** and efficient.

Scalability consideration

The program has a time complexity of O(n), which is efficient for small to medium-sized datasets. The usage of a HashMap data structure allows fast access to any group. However, as the number of words increases, performance may degrade. A better approach would be to leverage multiple **execution threads** to parallelize the computation. By dividing the input and distributing it across several threads, the total processing time can be significantly reduced.

To further optimize resource usage and avoid excessive thread creation overhead, it is recommended to use the **Thread Pool** pattern. This allows for better control over concurrency and improves memory efficiency.

Conclusion

Considering that the requirement did not involve a big dataset, the solution that I have provided is efficient. Although, if I needed to work with big data, I would apply the modifications described in the previous paragraph. As the program is modular, only the AnagramFinder.java class would be modified. The number of threads can be set according to the number of available processors at runtime.