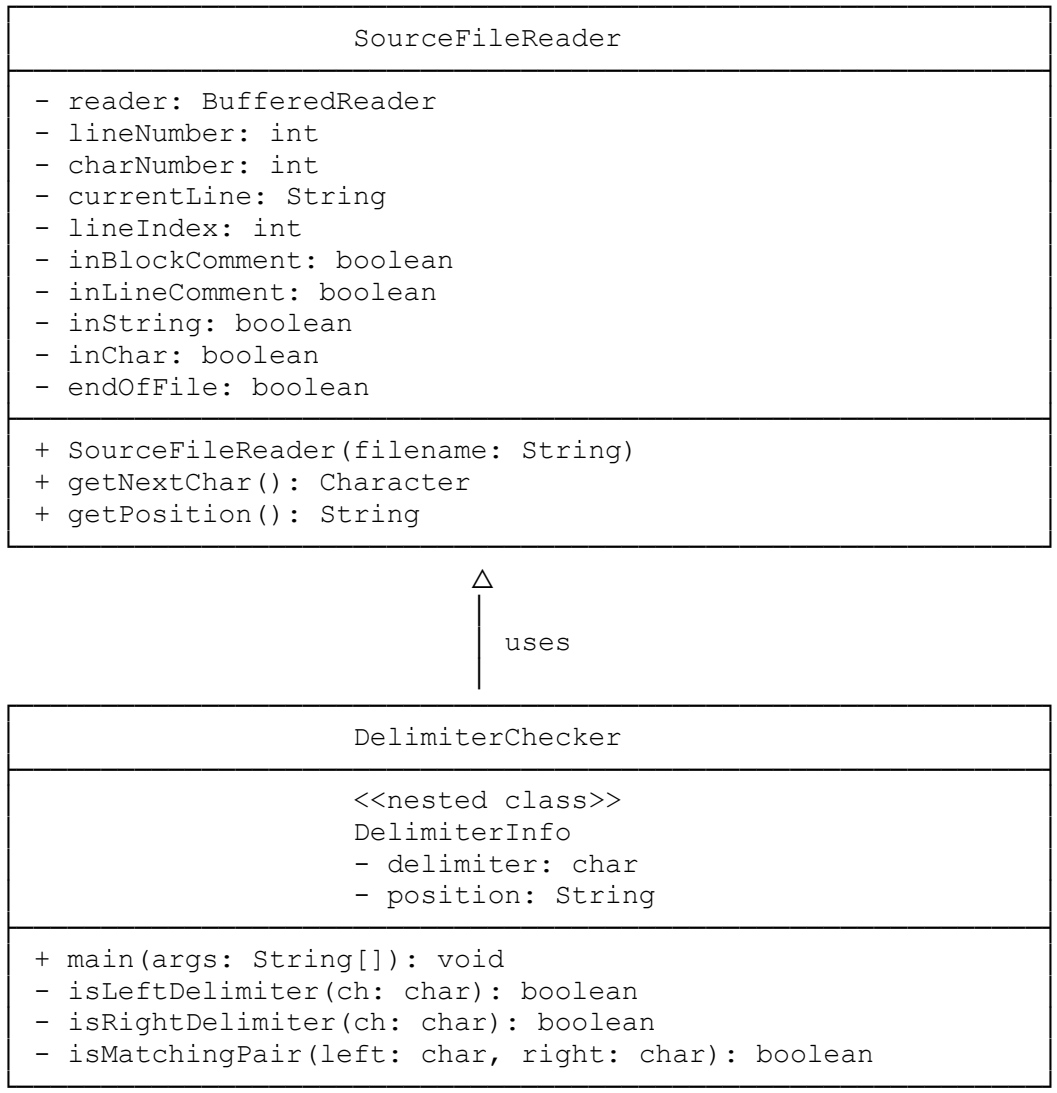
CMSC 315 – Project 1 Documentation
Java Delimiter Matching Program

Student: Stefan V. Nikolov
Date: January 20, 2026
Course: CMSC 315 – Data Structures and Analysis

---

1. UML Class Diagram



Class Relationships
- **DelimiterChecker** uses **SourceFileReader** to read and filter characters from the input file
- **DelimiterChecker** contains a nested helper class **DelimiterInfo** to store delimiter information with position tracking

- **SourceFileReader** encapsulates all file I/O and state tracking logic
- **DelimiterChecker** implements the main algorithm using Java's Stack class

---

2. Test Plan

Test Case 1: Valid Delimiter Matching
Purpose: Verify that properly matched delimiters are recognized as valid

Test File Content:
```java
public class Test {
    public static void main(String[] args) {
        int[] array = {1, 2, 3};
        System.out.println(array[0]);
    }
}
```

Expected Output:
```
All delimiters match correctly.
```

Status: Pass

Test Case 2: Unmatched Right Delimiter
Purpose: Detect when a closing delimiter appears without a corresponding opening delimiter

Test File Content:
```java
public class Test {
    public void method() {
        System.out.println("test");
    }}
}
```

Expected Output:
```
Unmatched delimiter '}' at Line 4, Character 5
```

Status: Pass

Test Case 3: Mismatched Delimiter Pair

Purpose: Detect when a closing delimiter doesn't match the most recent opening delimiter

Test File Content:
```java
public class Test {
    public void method() {
        if (true) {
            System.out.println("test");
        )
    }
}
```

Expected Output:
```
Mismatched delimiters '{' and ')' at Line 5, Character 9
```

Status: Pass

Test Case 4: Unmatched Left Delimiter at EOF
Purpose: Detect when an opening delimiter has no corresponding closing delimiter

Test File Content:
```java
public class Test {
    public void method() {
        System.out.println("test");
    }
// Missing closing brace
```

Expected Output:
```
Unmatched delimiter '{' at end of file.
```

Status: Pass

Test Case 5: Delimiters in Line Comments
Purpose: Verify that delimiters inside single-line comments are ignored

Test File Content:
```java
public class Test {
    // This comment has unmatched delimiters: { [ (
    public void method() {
```

```
        System.out.println("test");
    }
}
```

Expected Output:
```
All delimiters match correctly.
```

Status: Pass

Test Case 6: Delimiters in Block Comments
Purpose: Verify that delimiters inside block comments are ignored

Test File Content:
```java
public class Test {
    /* This block comment has
       unmatched delimiters: { [ ( } ] )
       and should be completely ignored */
    public void method() {
        System.out.println("test");
    }
}
```

Expected Output:
```
All delimiters match correctly.
```

Status: Pass

Test Case 7: Delimiters in String Literals
Purpose: Verify that delimiters inside string literals are ignored

Test File Content:
```java
public class Test {
    public void method() {
        String s = "This string has delimiters: {[(]})";
        System.out.println(s);
    }
}
```

Expected Output:
```
All delimiters match correctly.
```

```

**Status:** ✅ Pass

---

### Test Case 8: Delimiters in Character Literals
**Purpose:** Verify that delimiters inside character literals are ignored

**Test File Content:**
```java
public class Test {
    public void method() {
        char open = '(';
        char close = ')';
        char brace = '{';
    }
}
```

Expected Output:
```
All delimiters match correctly.
```

Status: ✅ Pass

Test Case 9: Mixed Nested Delimiters
Purpose: Verify correct handling of multiple types of nested delimiters

Test File Content:
```java
public class Test {
    public void method() {
        int[][] matrix = {{1, 2}, {3, 4}};
        if (matrix[0][0] == 1) {
            System.out.println("Valid");
        }
    }
}
```

Expected Output:
```
All delimiters match correctly.
```

Status: ✅ Pass

```

Test Case 10: Escaped Characters in Strings
Purpose: Verify that escaped quotes in strings are handled correctly

Test File Content:
```java
public class Test {
    public void method() {
        String s = "This has an escaped quote: \" and delimiter {";
        System.out.println(s);
    }
}
```

Expected Output:
```
All delimiters match correctly.
```

Status: Pass

Test Case 11: Empty File
Purpose: Verify handling of empty input files

Test File Content:
```
(empty file)
```

Expected Output:
```
All delimiters match correctly.
```

Status: Pass

Test Case 12: Multiple Errors (First Error Reported)
Purpose: Verify that the program stops at the first error encountered

Test File Content:
```java
public class Test {
    public void method() {
        System.out.println("test");
    )]
}
```

Expected Output:
```
Mismatched delimiters '{' and ']' at Line 4, Character 6
```

```

**Status:**  Pass

---

## 3. Lessons Learned

This project reinforced several fundamental concepts in data structures and software design. First, the importance of **state-driven parsing** became evident when implementing the character filtering logic—tracking multiple boolean flags simultaneously (for comments, strings, and character literals) required careful attention to state transitions and edge cases like escaped characters.

Second, the project demonstrated the practical utility of the **stack data structure** for delimiter matching, showing how LIFO (Last-In-First-Out) behavior naturally models nested structures in programming languages.

Third, the value of **separation of concerns** was clear: by encapsulating file reading logic in `SourceFileReader` and validation logic in `DelimiterChecker`, the code became more maintainable and testable. Each class had a single, well-defined responsibility.

Finally, the project highlighted the complexity of **incremental parsing**—reading character-by-character while maintaining accurate position tracking and handling multi-character sequences (like `//` and `/*`) requires meticulous logic. Edge cases such as escaped quotes, nested delimiters, and end-of-file conditions proved that thorough testing is essential for robust parser implementation. These lessons will be invaluable for future projects involving file processing, parsing, and data structure applications.

---

## 4. Conclusion

The Java Delimiter Matching Program successfully validates delimiter matching in Java source files while correctly handling comments, strings, and character literals. The implementation demonstrates practical application of stack-based algorithms and state-driven parsing techniques taught in CMSC 315.

---

**End of Documentation**
```