```java
import components.queue.Queue;
import components.simplereader.SimpleReader;
import components.simplereader.SimpleReader1L;
import components.simplewriter.SimpleWriter;
import components.simplewriter.SimpleWriter1L;
import components.statement.Statement;
import components.statement.Statement1;
import components.utilities.Reporter;
import components.utilities.Tokenizer;

/**
 * Layered implementation of secondary methods {@code parse} and
 * {@code parseBlock} for {@code Statement}.
 *
 * @author Chloe Feller and Krish Patel
 *
 */
public final class Statement1Parse1 extends Statement1 {

    /*
     * Private members --------------------------------------------------------
     */

    /**
     * Converts {@code c} into the corresponding {@code Condition}.
     *
     * @param c
     *            the condition to convert
     * @return the {@code Condition} corresponding to {@code c}
     * @requires [c is a condition string]
     * @ensures parseCondition = [Condition corresponding to c]
     */
    private static Condition parseCondition(String c) {
        assert c != null : "Violation of: c is not null";
        assert Tokenizer
                .isCondition(c) : "Violation of: c is a condition string";
        return Condition.valueOf(c.replace('-', '_').toUpperCase());
    }

    /**
     * Parses an IF or IF_ELSE statement from {@code tokens} into {@code s}.
     *
     * @param tokens
     *            the input tokens
     * @param s
     *            the parsed statement
     * @replaces s
     * @updates tokens
     * @requires <pre>
     * [<"IF"> is a prefix of tokens]  and
     *  [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
     * </pre>
     * @ensures <pre>
     * if [an if string is a proper prefix of #tokens] then
     *  s = [IF or IF_ELSE Statement corresponding to if string at start of #tokens]  and
     *  #tokens = [if string at start of #tokens] * tokens
     * else
     *  [reports an appropriate error message to the console and terminates client]
     * </pre>
     */
    private static void parseIf(Queue<String> tokens, Statement s) {
        assert tokens != null : "Violation of: tokens is not null";
        assert s != null : "Violation of: s is not null";
        assert tokens.length() > 0 && tokens.front().equals("IF") : ""
                + "Violation of: <\"IF\"> is proper prefix of tokens";

        /**
         * Check the keyword and condition.
         */
        String start = tokens.dequeue();
        Reporter.assertElseFatalError(Tokenizer.isKeyword(start),
                "Invalid token");

        String condition = tokens.dequeue();
        Reporter.assertElseFatalError(
                tokens.length() > 0 && Tokenizer.isCondition(condition),
                "Invalid condition");
        Condition c = parseCondition(condition);

        Reporter.assertElseFatalError(tokens.dequeue().equals("THEN"),
                "Invalid token");

        /**
         * Parse and assemble "IF" or "IF_ELSE".
         */
        Statement s1 = s.newInstance();
        s1.parseBlock(tokens);
        if (tokens.front().equals("ELSE")) {
            tokens.dequeue();
            Statement s2 = s.newInstance();
            s2.parseBlock(tokens);
            s.assembleIfElse(c, s1, s2);
        } else {
            s.assembleIf(c, s1);
        }

        /**
         * Check the end.
         */
        Reporter.assertElseFatalError(tokens.dequeue().equals("END"),
                "Invalid token");
        Reporter.assertElseFatalError(Tokenizer.isKeyword(tokens.dequeue()),
                "Invalid token");
    }

    /**
     * Parses a WHILE statement from {@code tokens} into {@code s}.
     *
     * @param tokens
     *            the input tokens
     * @param s
     *            the parsed statement
     * @replaces s
     * @updates tokens
     * @requires <pre>
     * [<"WHILE"> is a prefix of tokens]  and
     *  [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
     * </pre>
     * @ensures <pre>
     * if [a while string is a proper prefix of #tokens] then
     *  s = [WHILE Statement corresponding to while string at start of #tokens]  and
     *  #tokens = [while string at start of #tokens] * tokens
     * else
     *  [reports an appropriate error message to the console and terminates client]
     * </pre>
     */
    private static void parseWhile(Queue<String> tokens, Statement s) {
        assert tokens != null : "Violation of: tokens is not null";
        assert s != null : "Violation of: s is not null";
        assert tokens.length() > 0 && tokens.front().equals("WHILE") : ""
                + "Violation of: <\"WHILE\"> is proper prefix of tokens";

        String start = tokens.dequeue();
        Reporter.assertElseFatalError(Tokenizer.isKeyword(start),
                "Invalid token");

        Reporter.assertElseFatalError(Tokenizer.isCondition(tokens.front()),
                "Invalid condition");
        Condition c = parseCondition(tokens.dequeue());

        Reporter.assertElseFatalError(tokens.dequeue().equals("DO"),
                "Invalid token");

        Statement s1 = s.newInstance();
        s1.parseBlock(tokens);
        s.assembleWhile(c, s1);

        Reporter.assertElseFatalError(start.equals("WHILE"),
                "expecting 'WHILE'");
        start = tokens.dequeue();

        Reporter.assertElseFatalError(start.equals("END"), "Invalid token");
        Reporter.assertElseFatalError(Tokenizer.isKeyword(tokens.dequeue()),
                "Invalid token");

    }

    /**
     * Parses a CALL statement from {@code tokens} into {@code s}.
     *
     * @param tokens
     *            the input tokens
     * @param s
     *            the parsed statement
     * @replaces s
     * @updates tokens
     * @requires [identifier string is a proper prefix of tokens]
     * @ensures <pre>
     * s =
     *   [CALL Statement corresponding to identifier string at start of #tokens]  and
     *  #tokens = [identifier string at start of #tokens] * tokens
     * </pre>
     */
    private static void parseCall(Queue<String> tokens, Statement s) {
        assert tokens != null : "Violation of: tokens is not null";
        assert s != null : "Violation of: s is not null";
        assert tokens.length() > 0
                && Tokenizer.isIdentifier(tokens.front()) : ""
                        + "Violation of: identifier string is proper prefix of tokens";

        String name = tokens.dequeue();
        Reporter.assertElseFatalError(Tokenizer.isIdentifier(name),
                "Invalid token");
        s.assembleCall(name);
    }

    /*
     * Constructors -----------------------------------------------------------
     */

    /**
     * No-argument constructor.
     */
    public Statement1Parse1() {
        super();
    }

    /*
     * Public methods ---------------------------------------------------------
     */

    @Override
    public void parse(Queue<String> tokens) {
        assert tokens != null : "Violation of: tokens is not null";
        assert tokens.length() > 0 : ""
                + "Violation of: Tokenizer.END_OF_INPUT is a suffix of tokens";

        String token = tokens.front();

        Reporter.assertElseFatalError(
                Tokenizer.isIdentifier(token) || Tokenizer.isKeyword(token),
                "Invalid token");

        if (token.equals("WHILE")) {
            parseWhile(tokens, this);
        } else if (token.equals("IF")) {
            parseIf(tokens, this);
        } else {
            parseCall(tokens, this);
        }

    }

    @Override
    public void parseBlock(Queue<String> tokens) {
        assert tokens != null : "Violation of: tokens is not null";
        assert tokens.length() > 0 : ""
                + "Violation of: Tokenizer.END_OF_INPUT is a suffix of tokens";
```

```java

        this.clear();

        String token = tokens.front();

        for (int i = 0; Tokenizer.isIdentifier(token) || token.equals("IF")
                || token.equals("WHILE"); i++) {
            Statement s = this.newInstance();

            s.parse(tokens);
            this.addToBlock(i, s);

            token = tokens.front();
        }

    }

    /*
     * Main test method -------------------------------------------------------
     */

    /**
     * Main method.
     *
     * @param args
     *            the command line arguments
     */
    public static void main(String[] args) {
        SimpleReader in = new SimpleReader1L();
        SimpleWriter out = new SimpleWriter1L();
        /*
         * Get input file name
         */
        out.print("Enter valid BL statement(s) file name: ");
        String fileName = in.nextLine();
        /*
         * Parse input file
         */
        out.println("*** Parsing input file ***");
        Statement s = new Statement1Parse1();
        SimpleReader file = new SimpleReader1L(fileName);
        Queue<String> tokens = Tokenizer.tokens(file);
        file.close();
        s.parse(tokens); // replace with parseBlock to test other method
        /*
         * Pretty print the statement(s)
         */
        out.println("*** Pretty print of parsed statement(s) ***");
        s.prettyPrint(out, 0);

        in.close();
        out.close();
    }

}

```