```java
import components.sequence.Sequence;

/**
 * {@code Statement} represented as a {@code Tree<StatementLabel>} with
 * implementations of primary methods.
 *
 * @convention [$this.rep is a valid representation of a Statement]
 * @correspondence this = $this.rep
 *
 * @author Zhuoyang Li + Xinci Ma
 *
 */
public class Statement2 extends StatementSecondary {

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

    /**
     * Label class for the tree representation.
     */
    private static final class StatementLabel {

        /**
         * Statement kind.
         */
        private Kind kind;

        /**
         * IF/IF_ELSE/WHILE statement condition.
         */
        private Condition condition;

        /**
         * CALL instruction name.
         */
        private String instruction;

        /**
         * Constructor for BLOCK.
         *
         * @param k
         *            the kind of statement
         *
         * @requires k = BLOCK
         * @ensures this = (BLOCK, ?, ?)
         */
        private StatementLabel(Kind k) {
            assert k == Kind.BLOCK : "Violation of: k = BLOCK";
            this.kind = k;
        }

        /**
         * Constructor for IF, IF_ELSE, WHILE.
         *
         * @param k
         *            the kind of statement
         * @param c
         *            the statement condition
         *
         * @requires k = IF or k = IF_ELSE or k = WHILE
         * @ensures this = (k, c, ?)
         */
        private StatementLabel(Kind k, Condition c) {
            assert k == Kind.IF || k == Kind.IF_ELSE || k == Kind.WHILE : ""
                    + "Violation of: k = IF or k = IF_ELSE or k = WHILE";
            this.kind = k;
            this.condition = c;
        }

        /**
         * Constructor for CALL.
         *
         * @param k
         *            the kind of statement
         * @param i
         *            the instruction name
         *
         * @requires k = CALL and [i is an IDENTIFIER]
         * @ensures this = (CALL, ?, i)
         */
        private StatementLabel(Kind k, String i) {
            assert k == Kind.CALL : "Violation of: k = CALL";
            assert i != null : "Violation of: i is not null";
            assert Tokenizer
                    .isIdentifier(i) : "Violation of: i is an IDENTIFIER";
            this.kind = k;
            this.instruction = i;
        }

        @Override
        public String toString() {
            String condition = "?", instruction = "?";
            if ((this.kind == Kind.IF) || (this.kind == Kind.IF_ELSE)
                    || (this.kind == Kind.WHILE)) {
                condition = this.condition.toString();
            } else if (this.kind == Kind.CALL) {
                instruction = this.instruction;
            }
            return "(" + this.kind + "," + condition + "," + instruction + ")";
        }

    }

    /**
     * The tree representation field.
     */
    private Tree<StatementLabel> rep;

    /**
     * Creator of initial representation.
     */
    private void createNewRep() {

        this.rep = new Tree1<>();
        StatementLabel rootLabel = new StatementLabel(Kind.BLOCK);
        Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
        this.rep.assemble(rootLabel, children);

    }

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

    /**
     * No-argument constructor.
     */
    public Statement2() {
```

```java
        this.createNewRep();
    }

    /*
     * Standard methods -------------------------------------------------------
     */

    @Override
    public final Statement2 newInstance() {
        try {
            return this.getClass().getConstructor().newInstance();
        } catch (ReflectiveOperationException e) {
            throw new AssertionError(
                    "Cannot construct object of type " + this.getClass());
        }
    }

    @Override
    public final void clear() {
        this.createNewRep();
    }

    @Override
    public final void transferFrom(Statement source) {
        assert source != null : "Violation of: source is not null";
        assert source != this : "Violation of: source is not this";
        assert source instanceof Statement2 : ""
                + "Violation of: source is of dynamic type Statement2";
        /*
         * This cast cannot fail since the assert above would have stopped
         * execution in that case: source must be of dynamic type Statement2.
         */
        Statement2 localSource = (Statement2) source;
        this.rep = localSource.rep;
        localSource.createNewRep();
    }

    /*
     * Kernel methods
     */

    @Override
    public final Kind kind() {

        return this.rep.root().kind;
    }

    @Override
    public final void addToBlock(int pos, Statement s) {
        assert s != null : "Violation of: s is not null";
        assert s != this : "Violation of: s is not this";
        assert s instanceof Statement2 : "Violation of: s is a Statement2";
        assert this.kind() == Kind.BLOCK : ""
                + "Violation of: [this is a BLOCK statement]";
        assert 0 <= pos : "Violation of: 0 <= pos";
        assert pos <= this.lengthOfBlock() : ""
                + "Violation of: pos <= [length of this BLOCK]";
        assert s.kind() != Kind.BLOCK : "Violation of: [s is not a BLOCK statement]";

        Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
        Statement2 localS = (Statement2) s;
        StatementLabel label = this.rep.disassemble(children);
        children.add(pos, localS.rep);
        this.rep.assemble(label, children);
        localS.createNewRep(); // clear s

    }

    @Override
    public final Statement removeFromBlock(int pos) {
        assert 0 <= pos : "Violation of: 0 <= pos";
        assert pos < this.lengthOfBlock() : ""
                + "Violation of: pos < [length of this BLOCK]";
        assert this.kind() == Kind.BLOCK : ""
                + "Violation of: [this is a BLOCK statement]";
        /*
         * The following call to Statement newInstance method
```

```java
is a violation of
         * the kernel purity rule. However, there is no way to avoid it and it
         * is safe because the convention clearly holds at this point in the
         * code.
         */
        Statement2 s = this.newInstance();

        Tree<StatementLabel> removedSubtree = this.rep.removeSubtree(pos);
        Statement2 removedStatement = new Statement2();
        removedStatement.rep = removedSubtree;
        return removedStatement;
    }

    @Override
    public final int lengthOfBlock() {
        assert this.kind() == Kind.BLOCK : ""
                + "Violation of: [this is a BLOCK statement]";

        return this.rep.numberOfSubtrees();
    }

    @Override
    public final void assembleIf(Condition c, Statement s) {
        assert c != null : "Violation of: c is not null";
        assert s != null : "Violation of: s is not null";
        assert s != this : "Violation of: s is not this";
        assert s instanceof Statement2 : "Violation of: s is a Statement2";
        assert s.kind() == Kind.BLOCK : ""
                + "Violation of: [s is a BLOCK statement]";
        Statement2 localS = (Statement2) s;
        StatementLabel label = new StatementLabel(Kind.IF, c);
        Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
        children.add(0, localS.rep);
        this.rep.assemble(label, children);
        localS.createNewRep(); // clears s
    }

    @Override
    public final Condition disassembleIf(Statement s) {
        assert s != null : "Violation of: s is not null";
        assert s != this : "Violation of: s is not this";
        assert s instanceof Statement2 : "Violation of: s is a Statement2";
        assert this.kind() == Kind.IF : ""
                + "Violation of: [this is an IF statement]";
        Statement2 localS = (Statement2) s;
        Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
        StatementLabel label = this.rep.disassemble(children);
        localS.rep = children.remove(0);
        this.createNewRep(); // clears this
        return label.condition;
    }

    @Override
    public final void assembleIfElse(Condition c, Statement s1, Statement s2) {
        assert c != null : "Violation of: c is not null";
        assert s1 != null : "Violation of: s1 is not null";
        assert s2 != null : "Violation of: s2 is not null";
        assert s1 != this : "Violation of: s1 is not this";
        assert s2 != this : "Violation of: s2 is not this";
        assert s1 != s2 : "Violation of: s1 is not s2";
        assert s1 instanceof Statement2 : "Violation of: s1 is a Statement2";
        assert s2 instanceof Statement2 : "Violation of: s2 is a Statement2";
        assert s1
                .kind() == Kind.BLOCK : "Violation of: [s1 is a BLOCK statement]";
        assert s2
                .kind() == Kind.BLOCK : "Violation of: [s2 is a BLOCK statement]";

        Statement2 thenStatement = (Statement2) s1;
        Statement2 elseStatement = (Statement2) s2;
        Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();

        children.add(0, thenStatement.rep);
        children.add(1, elseStatement.rep);
        this.rep.assemble(new StatementLabel(Kind.IF_ELSE, c), children);

        thenStatement.createNewRep(); // clear input statement
        elseStatement.createNewRep();

    }

    @Override
    public final Condition disassembleIfElse(Statement s1, Statement s2) {
        assert s1 != null : "Violation of: s1 is not null";
        assert s2 != null : "Violation of: s1 is not null";
        assert s1 != this : "Violation of: s1 is not this";
        assert s2 != this : "Violation of: s2 is not this";
        assert s1 != s2 : "Violation of: s1 is not s2";
        assert s1 instanceof Statement2 : "Violation of: s1 is a Statement2";
        assert s2 instanceof Statement2 : "Violation of: s2 is a Statement2";
        assert this.kind() == Kind.IF_ELSE : ""
                + "Violation of: [this is an IF_ELSE statement]";

        Statement2 thenStatement = (Statement2) s1;
        Statement2 elseStatement = (Statement2) s2;
        Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();

        StatementLabel label = this.rep.disassemble(children);
        thenStatement.rep = children.remove(0);
        elseStatement.rep = children.remove(0);
        this.createNewRep();

        return label.condition;
    }

    @Override
    public final void assembleWhile(Condition c, Statement s) {
        assert c != null : "Violation of: c is not null";
        assert s != null : "Violation of: s is not null";
        assert s != this : "Violation of: s is not this";
        assert s instanceof Statement2 : "Violation of: s is a Statement2";
        assert s.kind() == Kind.BLOCK : "Violation of: [s is a BLOCK statement]";

        // casting s to Statement2 to work with representation
        Statement2 sAsStatement2 = (Statement2) s;

        // creating new sequence for children of WHILE statement
        Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();

        // adding representation of s as only child of WHILE statement
        children.add(0, sAsStatement2.rep);

        // assembling WHILE statement with condition and single child
        this.rep.assemble(new StatementLabel(Kind.WHILE, c), children);

        // clearing original statement s to ensure solely part of WHILE structure
        sAsStatement2.createNewRep();
    }

    @Override
    public final Condition disassembleWhile(Statement s) {
        assert s != null : "Violation of: s is not null";
        assert s != this : "Violation of: s is not this";
        assert s instanceof Statement2 : "Violation of: s is a Statement2";
        assert this.kind() == Kind.WHILE : ""
                + "Violation of: [this is a WHILE statement]";

        // preparing to extract children (body) of WHILE statement
        Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();

        // extracting label (contains condition) and body
        StatementLabel label = this.rep.disassemble(children);

        // casting s to Statement2 to modify to represent body of WHILE
        Statement2 bodyStatement = (Statement2) s;

        // assuming WHILE has one body statement
        bodyStatement.rep = children.remove(0);
        this.createNewRep();

        // returning condition part of WHILE statement
        return label.condition;
    }

    @Override
    public final void assembleCall(String inst) {
        assert inst != null : "Violation of: inst is not null";
        assert Tokenizer.isIdentifier(inst) : ""
                + "Violation of: inst is a valid IDENTIFIER";

        // create label for CALL with instruction name
        StatementLabel label = new StatementLabel(Kind.CALL, inst);

        // as CALL statements have no children, create empty sequence for children
        Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();

        // assembling CALL statement with label and no children
        this.rep.assemble(label, children);

    }

    @Override
    public final String disassembleCall() {
        assert this.kind() == Kind.CALL : ""
                + "Violation of: [this is a CALL statement]";

        // preparing to extract potential children
        Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();

        // extracting label which contains the instruction name
        StatementLabel label = this.rep.disassemble(children);

        // clearing representation to return only instruction name
        this.createNewRep();

        // returning instruction name part of CALL statement
        return label.instruction;
    }

}

```