JStringConcatenationOp.java

```java
// Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a string concatenation operation. Nodes of this type are not
 * produced by the parser but by analysis of a + operation where the arguments
 * are strings. Such operations are rewritten to be string concatenation
 * operations.
 */

class JStringConcatenationOp extends JBinaryExpression {

    /**
     * Construct an AST node for a string concatenation expression given its
     * line number, and the lhs and rhs operands. An expression of this sort is
     * created during the analysis of a (overloaded) + operation (and not by the
     * Parser).
     * 
     * @param line
     *            line in which the expression occurs in the source file.
     * @param lhs
     *            lhs operand.
     * @param rhs
     *            rhs operand.
     */

    public JStringConcatenationOp(int line, JExpression lhs, JExpression rhs) {
        super(line, "+", lhs, rhs);
    }

    /**
     * Analysis is simple here. The operands have already been analyzed (in
     * JPlusOp) so we simply set the result type.
     * 
     * @param context
     *            context in which names are resolved.
     * @return the analyzed (and possibly rewritten) AST subtree.
     */

    public JExpression analyze(Context context) {
        type = Type.STRING;
        return this;
    }

    /**
     * Code generation generates code for creating a StringBuilder atop the
     * runtime stack, appending the operands (which might contain nested
     * concatenations; these are handled by cascadingCodegen()), and then for
     * converting the StringBuilder to a String.
     * 
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    public void codegen(CLEmitter output) {
        // Firstly, create a StringBuilder
        output.addReferenceInstruction(NEW, "java/lang/StringBuilder");
        output.addNoArgInstruction(DUP);
        output.addMemberAccessInstruction(INVOKESPECIAL,
                "java/lang/StringBuilder", "<init>", "()V");

        // Lhs and Rhs
        nestedCodegen(output);

        // Finally, make into a String
        output.addMemberAccessInstruction(INVOKEVIRTUAL,
                "java/lang/StringBuilder", "toString", "()Ljava/lang/String;");
    }

    /**
     * Like a codegen() but we needn't (and shouldn't) create a StringBuilder
     * nor convert the result to a String, as that will be done in a parent.
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    void nestedCodegen(CLEmitter output) {
        // Lhs
        if (lhs instanceof JStringConcatenationOp) {
            // This appends lhs
            ((JStringConcatenationOp) lhs).nestedCodegen(output);
        } else {
            lhs.codegen(output);
            output.addMemberAccessInstruction(INVOKEVIRTUAL,
                    "java/lang/StringBuilder", "append", "("
                            + lhs.type().argumentTypeForAppend()
                            + ")Ljava/lang/StringBuilder;");
        }

        // Rhs
        if (rhs instanceof JStringConcatenationOp) {
            // This appends rhs
            ((JStringConcatenationOp) rhs).nestedCodegen(output);
        } else {
            rhs.codegen(output);
            output.addMemberAccessInstruction(INVOKEVIRTUAL,
                    "java/lang/StringBuilder", "append", "("
                            + rhs.type().argumentTypeForAppend()
                            + ")Ljava/lang/StringBuilder;");
        }
    }

}
```