JReturnStatement.java

```java
// Copyright 2011 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a return-statement. If the enclosing method
 * in non-void, then there is a value to return, so we keep track
 * of the expression denoting that value and its type.
 */

class JReturnStatement
    extends JStatement {

    /** The returned expression. */
    private JExpression expr;

    /**
     * Construct an AST node for a return-statement given its
     * line number, and the expression that is returned.
     * 
     * @param line
     *                line in which the return-statement appears
     *                in the source file.
     * @param expr
     *                the returned expression.
     */

    public JReturnStatement(int line, JExpression expr) {
        super(line);
        this.expr = expr;
    }

    /**
     * Analysis distinguishes between our being in a constructor
     * or in a regular method in checking return types. In the
     * case of a return expression, analyze it and check types.
     * Determine the (possibly void) return type.
     * 
     * @param context
     *                context in which names are resolved.
     * @return the analyzed (and possibly rewritten) AST subtree.
     */

    public JStatement analyze(Context context) {
        MethodContext methodContext = context.methodContext();

        // The methodContext can be null if return statement
        // occurs
        // in a block that is not within a method. For example,
        // in
        // the Java grammar, return statement, at least
        // syntactically, can occur in a static block. But since
        // j-- does not allow a block to occur outside of a
        // method,
        // we don't check for methodContext being null

        if (methodContext.methodReturnType() == Type.CONSTRUCTOR) {
            if (expr != null) {
                // Can't return a value from a constructor
                JAST.compilationUnit.reportSemanticError(line(),
                    "cannot return a value from a constructor");
            }
        } else {
            // Must be a method
            Type returnType = methodContext.methodReturnType();
        methodContext.confirmMethodHasReturn();
            if (expr != null) {
                if (returnType == Type.VOID) {
                    // Can't return a value from void method
                    JAST.compilationUnit.reportSemanticError(line(),
                        "cannot return a value from a void method");
                } else {
                    // There's a (non-void) return expression.
                    // Its
                    // type must match the return type of the
                    // method
                    expr = expr.analyze(context);
                    expr.type().mustMatchExpected(line(), returnType);
                }
            } else {
                // The method better have void as return type
                if (returnType != Type.VOID) {
                    JAST.compilationUnit.reportSemanticError(line(),
                        "missing return value");
                }
            }
        }
        return this;
    }

    /**
     * Generate code for the return statement. In the case of
     * void method types, generate a simple (void) return. In the
     * case of a return expression, generate code to load that
     * onto the stack and then generate the appropriate return
     * instruction.
     *
     * @param output
     *                the code emitter (basically an abstraction
     *                for producing the .class file).
     */

    public void codegen(CLEmitter output) {
        if (expr == null) {
            output.addNoArgInstruction(RETURN);
        } else {
            expr.codegen(output);
            if (expr.type() == Type.INT
                || expr.type() == Type.BOOLEAN
                || expr.type() == Type.CHAR) {
                output.addNoArgInstruction(IRETURN);
            } else {
                output.addNoArgInstruction(ARETURN);
            }
        }
    }

    /**
     * @inheritDoc
     */

    public void writeToStdOut(PrettyPrinter p) {
        if (expr != null) {
            p.printf("<JReturnStatement line=\"%d\">\n", line());
            p.indentRight();
            expr.writeToStdOut(p);
            p.indentLeft();
            p.printf("</JReturnStatement>\n");
        } else {
            p.printf("<JReturnStatement line=\"%d\"/>\n", line());
        }
    }
}
```