JConstructorDeclaration.java

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

package jminusminus;

import java.util.ArrayList;
import static jminusminus.CLConstants.*;

/**
 * The AST node for a constructor declaration. A constructor looks very much
 * like a method.
 */

class JConstructorDeclaration extends JMethodDeclaration implements JMember {

    /** Does this constructor invoke this(...) or super(...)? */
    private boolean invokesConstructor;

    /** Defining class */
    JClassDeclaration definingClass;

    /**
     * Construct an AST node for a constructor declaration given the line
     * number, modifiers, constructor name, formal parameters, and the
     * constructor body.
     *
     * @param line
     *            line in which the constructor declaration occurs in the source
     *            file.
     * @param mods
     *            modifiers.
     * @param name
     *            constructor name.
     * @param params
     *            the formal parameters.
     * @param body
     *            constructor body.
     */

    public JConstructorDeclaration(int line, ArrayList<String> mods,
            String name, ArrayList<JFormalParameter> params, JBlock body)

    {
        super(line, mods, name, Type.CONSTRUCTOR, params, body);
    }

    /**
     * Declare this constructor in the parent (class) context.
     *
     * @param context
     *            the parent (class) context.
     * @param partial
     *            the code emitter (basically an abstraction for producing the
     *            partial class).
     */

    public void preAnalyze(Context context, CLEmitter partial) {
        super.preAnalyze(context, partial);
        if (isStatic) {
            JAST.compilationUnit.reportSemanticError(line(),
                    "Constructor cannot be declared static");
        } else if (isAbstract) {
            JAST.compilationUnit.reportSemanticError(line(),
                    "Constructor cannot be declared abstract");
        }
        if (body.statements().size() > 0
                && body.statements().get(0) instanceof JStatementExpression) {
```

```
            JStatementExpression first = (JStatementExpression) body
                    .statements().get(0);
            if (first.expr instanceof JSuperConstruction) {
                ((JSuperConstruction) first.expr).markProperUseOfConstructor();
                invokesConstructor = true;
            } else if (first.expr instanceof JThisConstruction) {
                ((JThisConstruction) first.expr).markProperUseOfConstructor();
                invokesConstructor = true;
            }
        }
    }

    /**
     * Analysis for a constructor declaration is very much like that for a
     * method declaration.
     *
     * @param context
     *            context in which names are resolved.
     * @return the analyzed (and possibly rewritten) AST subtree.
     */

    public JAST analyze(Context context) {
        // Record the defining class declaration.
        definingClass =
        (JClassDeclaration) context.classContext().definition();
        MethodContext methodContext =
            new MethodContext(context, isStatic, returnType);
        this.context = methodContext;

        if (!isStatic) {
            // Offset 0 is used to address "this".
            this.context.nextOffset();
        }

        // Declare the parameters. We consider a formal parameter
        // to be always initialized, via a function call.
        for (JFormalParameter param : params) {
            LocalVariableDefn defn =
        new LocalVariableDefn(param.type(),
                      this.context.nextOffset());
            defn.initialize();
            this.context.addEntry(param.line(), param.name(), defn);
        }
        if (body != null) {
            body = body.analyze(this.context);
        }
        return this;

    }

    /**
     * Add this constructor declaration to the partial class.
     *
     * @param context
     *            the parent (class) context.
     * @param partial
     *            the code emitter (basically an abstraction for producing the
     *            partial class).
     */

    public void partialCodegen(Context context, CLEmitter partial) {
        partial.addMethod(mods, "<init>", descriptor, null, false);
        if (!invokesConstructor) {
            partial.addNoArgInstruction(ALOAD_0);
            partial.addMemberAccessInstruction(INVOKESPECIAL,
                    ((JTypeDecl) context.classContext().definition())
                            .superType().jvmName(), "<init>", "()V");
        }
        partial.addNoArgInstruction(RETURN);
```

```java
    }

    /**
     * Generate code for the constructor declaration.
     * 
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    public void codegen(CLEmitter output) {
        output.addMethod(mods, "<init>", descriptor, null, false);
        if (!invokesConstructor) {
            output.addNoArgInstruction(ALOAD_0);
            output.addMemberAccessInstruction(INVOKESPECIAL,
                    ((JTypeDecl) context.classContext().definition())
                            .superType().jvmName(), "<init>", "()V");
        }
        // Field initializations
        for (JFieldDeclaration field : definingClass
                .instanceFieldInitializations()) {
            field.codegenInitializations(output);
        }
        // And then the body
        body.codegen(output);
        output.addNoArgInstruction(RETURN);
    }

    /**
     * @inheritDoc
     */

    public void writeToStdOut(PrettyPrinter p) {
        p.printf("<JConstructorDeclaration line=\"%d\" " + "name=\"%s\">\n",
                line(), name);
        p.indentRight();
        if (context != null) {
            context.writeToStdOut(p);
        }
        if (mods != null) {
            p.println("<Modifiers>");
            p.indentRight();
            for (String mod : mods) {
                p.printf("<Modifier name=\"%s\"/>\n", mod);
            }
            p.indentLeft();
            p.println("</Modifiers>");
        }
        if (params != null) {
            p.println("<FormalParameters>");
            for (JFormalParameter param : params) {
                p.indentRight();
                param.writeToStdOut(p);
                p.indentLeft();
            }
            p.println("</FormalParameters>");
        }
        if (body != null) {
            p.println("<Body>");
            p.indentRight();
            body.writeToStdOut(p);
            p.indentLeft();
            p.println("</Body>");
        }
        p.indentLeft();
        p.println("</JConstructorDeclaration>");
    }

}
```