JMethodDeclaration.java

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

package jminusminus;

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

/**
 * The AST node for a method declaration.
 */

class JMethodDeclaration
    extends JAST implements JMember {

    /** Method modifiers. */
    protected ArrayList<String> mods;

    /** Method name. */
    protected String name;

    /** Return type. */
    protected Type returnType;

    /** The formal parameters. */
    protected ArrayList<JFormalParameter> params;

    /** Method body. */
    protected JBlock body;

    /** Built in analyze(). */
    protected MethodContext context;

    /** Computed by preAnalyze(). */
    protected String descriptor;

    /** Is method abstract. */
    protected boolean isAbstract;

    /** Is method static. */
    protected boolean isStatic;

    /** Is method private. */
    protected boolean isPrivate;

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

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

    {
        super(line);
        this.mods = mods;
        this.name = name;
        this.returnType = returnType;
        this.params = params;
        this.body = body;
        this.isAbstract = mods.contains("abstract");
        this.isStatic = mods.contains("static");
        this.isPrivate = mods.contains("private");
    }

    /**
     * Declare this method 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) {
        // Resolve types of the formal parameters
        for (JFormalParameter param : params) {
            param.setType(param.type().resolve(context));
        }

        // Resolve return type
        returnType = returnType.resolve(context);

        // Check proper local use of abstract
        if (isAbstract && body != null) {
            JAST.compilationUnit.reportSemanticError(line(),
                "abstract method cannot have a body");
        } else if (body == null && !isAbstract) {
            JAST.compilationUnit.reportSemanticError(line(),
                "Method with null body must be abstarct");
        } else if (isAbstract && isPrivate) {
            JAST.compilationUnit.reportSemanticError(line(),
                "private method cannot be declared abstract");
        } else if (isAbstract && isStatic) {
            JAST.compilationUnit.reportSemanticError(line(),
                "static method cannot be declared abstract");
        }

        // Compute descriptor
        descriptor = "(";
        for (JFormalParameter param : params) {
            descriptor += param.type().toDescriptor();
        }
        descriptor += ")" + returnType.toDescriptor();

        // Generate the method with an empty body (for now)
        partialCodegen(context, partial);
    }

    /**
     * Analysis for a method declaration involves (1) creating a
     * new method context (that records the return type; this is
     * used in the analysis of the method body), (2) bumping up
     * the offset (for instance methods), (3) declaring the
     * formal parameters in the method context, and (4) analyzing
     * the method's body.
     *
     * @param context
     *               context in which names are resolved.
     * @return the analyzed (and possibly rewritten) AST subtree.
     */

    public JAST analyze(Context context) {
        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);
        if (returnType!=Type.VOID && ! methodContext.methodHasReturn()){
        JAST.compilationUnit.reportSemanticError(line(),
            "Non-void method must have a return statement");
        }
        }
    return this;
    }
```

```java
    /**
     * Add this method 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) {
        // Generate a method with an empty body; need a return to
        // make
        // the class verifier happy.
        partial.addMethod(mods, name, descriptor, null, false);

        // Add implicit RETURN
        if (returnType == Type.VOID) {
            partial.addNoArgInstruction(RETURN);
        } else if (returnType == Type.INT
            || returnType == Type.BOOLEAN || returnType == Type.CHAR) {
            partial.addNoArgInstruction(ICONST_0);
            partial.addNoArgInstruction(IRETURN);
        } else {
            // A reference type.
            partial.addNoArgInstruction(ACONST_NULL);
            partial.addNoArgInstruction(ARETURN);
        }
    }

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

    public void codegen(CLEmitter output) {
        output.addMethod(mods, name, descriptor, null, false);
        if (body != null) {
            body.codegen(output);
        }

        // Add implicit RETURN
        if (returnType == Type.VOID) {
            output.addNoArgInstruction(RETURN);
        }
    }

    /**
     * @inheritDoc
     */

    public void writeToStdOut(PrettyPrinter p) {
        p.printf("<JMethodDeclaration line=\"%d\" name=\"%s\" "
            + "returnType=\"%s\">\n", line(), name, returnType
            .toString());
        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("</JMethodDeclaration>");
    }
}
```