JClassDeclaration.java

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

package jminusminus;

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

/**
 * A class declaration has a list of modifiers, a name, a super class and a
 * class block; it distinguishes between instance fields and static (class)
 * fields for initialization, and it defines a type. It also introduces its own
 * (class) context.
 */

class JClassDeclaration extends JAST implements JTypeDecl {

    /** Class modifiers. */
    private ArrayList<String> mods;

    /** Class name. */
    private String name;

    /** Class block. */
    private ArrayList<JMember> classBlock;

    /** Super class type. */
    private Type superType;

    /** This class type. */
    private Type thisType;

    /** Context for this class. */
    private ClassContext context;

    /** Whether this class has an explicit constructor. */
    private boolean hasExplicitConstructor;

    /** Instance fields of this class. */
    private ArrayList<JFieldDeclaration> instanceFieldInitializations;

    /** Static (class) fields of this class. */
    private ArrayList<JFieldDeclaration> staticFieldInitializations;

    /**
     * Construct an AST node for a class declaration given the line number, list
     * of class modifiers, name of the class, its super class type, and the
     * class block.
     * 
     * @param line
     *            line in which the class declaration occurs in the source file.
     * @param mods
     *            class modifiers.
     * @param name
     *            class name.
     * @param superType
     *            super class type.
     * @param classBlock
     *            class block.
     */

    public JClassDeclaration(int line, ArrayList<String> mods, String name,
            Type superType, ArrayList<JMember> classBlock) {
        super(line);
        this.mods = mods;
        this.name = name;
        this.superType = superType;
```

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

```
        this.classBlock = classBlock;
        hasExplicitConstructor = false;
        instanceFieldInitializations = new ArrayList<JFieldDeclaration>();
        staticFieldInitializations = new ArrayList<JFieldDeclaration>();
    }

    /**
     * Return the class name.
     *
     * @return the class name.
     */

    public String name() {
        return name;
    }

    /**
     * Return the class' super class type.
     *
     * @return the super class type.
     */

    public Type superType() {
        return superType;
    }

    /**
     * Return the type that this class declaration defines.
     *
     * @return the defined type.
     */

    public Type thisType() {
        return thisType;
    }

    /**
     * The initializations for instance fields (now expressed as assignment
     * statments).
     *
     * @return the field declarations having initializations.
     */

    public ArrayList<JFieldDeclaration> instanceFieldInitializations() {
        return instanceFieldInitializations;
    }

    /**
     * Declare this class in the parent (compilation unit) context.
     *
     * @param context
     *            the parent (compilation unit) context.
     */

    public void declareThisType(Context context) {
        String qualifiedName = JAST.compilationUnit.packageName() == "" ? name
                : JAST.compilationUnit.packageName() + "/" + name;
        CLEmitter partial = new CLEmitter(false);
        partial.addClass(mods, qualifiedName, Type.OBJECT.jvmName(), null,
                false); // Object for superClass, just for now
        thisType = Type.typeFor(partial.toClass());
        context.addType(line, thisType);
    }

    /**
     * Pre-analyze the members of this declaration in the parent context.
     * Pre-analysis extends to the member headers (including method headers) but
     * not into the bodies.
     *
```

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

```
     * @param context
     *            the parent (compilation unit) context.
     */

    public void preAnalyze(Context context) {
        // Construct a class context
        this.context = new ClassContext(this, context);

        // Resolve superclass
        superType = superType.resolve(this.context);

        // Creating a partial class in memory can result in a
        // java.lang.VerifyError if the semantics below are
        // violated, so we can't defer these checks to analyze()
        thisType.checkAccess(line, superType);
        if (superType.isFinal()) {
            JAST.compilationUnit.reportSemanticError(line,
                    "Cannot extend a final type: %s", superType.toString());
        }

        // Create the (partial) class
        CLEmitter partial = new CLEmitter(false);

        // Add the class header to the partial class
        String qualifiedName = JAST.compilationUnit.packageName() == "" ? name
                : JAST.compilationUnit.packageName() + "/" + name;
        partial.addClass(mods, qualifiedName, superType.jvmName(), null, false);

        // Pre-analyze the members and add them to the partial
        // class
        for (JMember member : classBlock) {
            member.preAnalyze(this.context, partial);
            if (member instanceof JConstructorDeclaration
                    && ((JConstructorDeclaration) member).params.size() == 0) {
                hasExplicitConstructor = true;
            }
        }

        // Add the implicit empty constructor?
        if (!hasExplicitConstructor) {
            codegenPartialImplicitConstructor(partial);
        }

        // Get the Class rep for the (partial) class and make it
        // the
        // representation for this type
        Type id = this.context.lookupType(name);
        if (id != null && !JAST.compilationUnit.errorHasOccurred()) {
            id.setClassRep(partial.toClass());
        }
    }

    /**
     * Perform semantic analysis on the class and all of its members within the
     * given context. Analysis includes field initializations and the method
     * bodies.
     *
     * @param context
     *            the parent (compilation unit) context. Ignored here.
     * @return the analyzed (and possibly rewritten) AST subtree.
     */

    public JAST analyze(Context context) {
        // Analyze all members
        for (JMember member : classBlock) {
            ((JAST) member).analyze(this.context);
        }

        // Copy declared fields for purposes of initialization.
```

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

```java
        for (JMember member : classBlock) {
            if (member instanceof JFieldDeclaration) {
                JFieldDeclaration fieldDecl = (JFieldDeclaration) member;
                if (fieldDecl.mods().contains("static")) {
                    staticFieldInitializations.add(fieldDecl);
                } else {
                    instanceFieldInitializations.add(fieldDecl);
                }
            }
        }

        // Finally, ensure that a non-abstract class has
        // no abstract methods.
        if (!thisType.isAbstract() && thisType.abstractMethods().size() > 0) {
            String methods = "";
            for (Method method : thisType.abstractMethods()) {
                methods += "\n" + method;
            }
            JAST.compilationUnit.reportSemanticError(line,
                    "Class must be declared abstract since it defines "
                            + "the following abstract methods: %s", methods);

        }
        return this;
    }

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

    public void codegen(CLEmitter output) {
        // The class header
        String qualifiedName = JAST.compilationUnit.packageName() == "" ? name
                : JAST.compilationUnit.packageName() + "/" + name;
        output.addClass(mods, qualifiedName, superType.jvmName(), null, false);

        // The implicit empty constructor?
        if (!hasExplicitConstructor) {
            codegenImplicitConstructor(output);
        }

        // The members
        for (JMember member : classBlock) {
            ((JAST) member).codegen(output);
        }

        // Generate a class initialization method?
        if (staticFieldInitializations.size() > 0) {
            codegenClassInit(output);
        }
    }

    /**
     * @inheritDoc
     */

    public void writeToStdOut(PrettyPrinter p) {
        p.printf("<JClassDeclaration line=\"%d\" name=\"%s\""
                + " super=\"%s\">\n", line(), name, superType.toString());
        p.indentRight();
        if (context != null) {
            context.writeToStdOut(p);
        }
        if (mods != null) {
            p.println("<Modifiers>");
```

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

```
            p.indentRight();
            for (String mod : mods) {
                p.printf("<Modifier name=\"%s\"/>\n", mod);
            }
            p.indentLeft();
            p.println("</Modifiers>");
        }
        if (classBlock != null) {
            p.println("<ClassBlock>");
            for (JMember member : classBlock) {
                ((JAST) member).writeToStdOut(p);
            }
            p.println("</ClassBlock>");
        }
        p.indentLeft();
        p.println("</JClassDeclaration>");
    }

    /**
     * Generate code for an implicit empty constructor. (Necessary only if there
     * is not already an explicit one.)
     *
     * @param partial
     *            the code emitter (basically an abstraction for producing a
     *            Java class).
     */

    private void codegenPartialImplicitConstructor(CLEmitter partial) {
        // Invoke super constructor
        ArrayList<String> mods = new ArrayList<String>();
        mods.add("public");
        partial.addMethod(mods, "<init>", "()V", null, false);
        partial.addNoArgInstruction(ALOAD_0);
        partial.addMemberAccessInstruction(INVOKESPECIAL, superType.jvmName(),
                "<init>", "()V");

        // Return
        partial.addNoArgInstruction(RETURN);
    }

    /**
     * Generate code for an implicit empty constructor. (Necessary only if there
     * is not already an explicit one.
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    private void codegenImplicitConstructor(CLEmitter output) {
        // Invoke super constructor
        ArrayList<String> mods = new ArrayList<String>();
        mods.add("public");
        output.addMethod(mods, "<init>", "()V", null, false);
        output.addNoArgInstruction(ALOAD_0);
        output.addMemberAccessInstruction(INVOKESPECIAL, superType.jvmName(),
                "<init>", "()V");

        // If there are instance field initializations, generate
        // code for them
        for (JFieldDeclaration instanceField : instanceFieldInitializations) {
            instanceField.codegenInitializations(output);
        }

        // Return
        output.addNoArgInstruction(RETURN);
    }

    /**
```

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

```java
     * Generate code for class initialization, in j-- this means static field
     * initializations.
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    private void codegenClassInit(CLEmitter output) {
        ArrayList<String> mods = new ArrayList<String>();
        mods.add("public");
        mods.add("static");
        output.addMethod(mods, "<clinit>", "()V", null, false);

        // If there are instance initializations, generate code
        // for them
        for (JFieldDeclaration staticField : staticFieldInitializations) {
            staticField.codegenInitializations(output);
        }

        // Return
        output.addNoArgInstruction(RETURN);
    }

}
```

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder