

## JClassDeclaration.java

```
1  // Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3  package jminusminus;
4
5  import java.util.ArrayList;
6  import static jminusminus.CLConstants.*;
7
8  /**
9   * A class declaration has a list of modifiers, a name, a super class and a
10  * class block; it distinguishes between instance fields and static (class)
11  * fields for initialization, and it defines a type. It also introduces its own
12  * (class) context.
13  */
14
15  class JClassDeclaration extends JAST implements JTypeDecl {
16
17      /** Class modifiers. */
18      private ArrayList<String> mods;
19
20      /** Class name. */
21      private String name;
22
23      /** Class block. */
24      private ArrayList<JMember> classBlock;
25
26      /** Super class type. */
27      private Type superType;
28
29      /** This class type. */
30      private Type thisType;
31
32      /** Context for this class. */
33      private ClassContext context;
34
35      /** Whether this class has an explicit constructor. */
36      private boolean hasExplicitConstructor;
37
38      /** Instance fields of this class. */
39      private ArrayList<JFieldDeclaration> instanceFieldInitializations;
40
41      /** Static (class) fields of this class. */
42      private ArrayList<JFieldDeclaration> staticFieldInitializations;
43
44      /**
45       * Construct an AST node for a class declaration given the line number, list
46       * of class modifiers, name of the class, its super class type, and the
47       * class block.
48       *
49       * @param line
50       *         line in which the class declaration occurs in the source file.
51       * @param mods
52       *         class modifiers.
53       * @param name
54       *         class name.
55       * @param superType
56       *         super class type.
57       * @param classBlock
58       *         class block.
59       */
60
61      public JClassDeclaration(int line, ArrayList<String> mods, String name,
62                              Type superType, ArrayList<JMember> classBlock) {
63          super(line);
64          this.mods = mods;
65          this.name = name;
66          this.superType = superType;
```

```

67         this.classBlock = classBlock;
68         hasExplicitConstructor = false;
69         instanceFieldInitializations = new ArrayList<JFieldDeclaration>();
70         staticFieldInitializations = new ArrayList<JFieldDeclaration>();
71     }
72
73     /**
74      * Return the class name.
75      *
76      * @return the class name.
77      */
78
79     public String name() {
80         return name;
81     }
82
83     /**
84      * Return the class' super class type.
85      *
86      * @return the super class type.
87      */
88
89     public Type superType() {
90         return superType;
91     }
92
93     /**
94      * Return the type that this class declaration defines.
95      *
96      * @return the defined type.
97      */
98
99     public Type thisType() {
100         return thisType;
101     }
102
103     /**
104      * The initializations for instance fields (now expressed as assignment
105      * statements).
106      *
107      * @return the field declarations having initializations.
108      */
109
110     public ArrayList<JFieldDeclaration> instanceFieldInitializations() {
111         return instanceFieldInitializations;
112     }
113
114     /**
115      * Declare this class in the parent (compilation unit) context.
116      *
117      * @param context
118      *        the parent (compilation unit) context.
119      */
120
121     public void declareThisType(Context context) {
122         String qualifiedName = JAST.compilationUnit.packageName() == "" ? name
123             : JAST.compilationUnit.packageName() + "/" + name;
124         CLEmitter partial = new CLEmitter(false);
125         partial.addClass(mods, qualifiedName, Type.OBJECT.jvmName(), null,
126             false); // Object for superClass, just for now
127         thisType = Type.typeFor(partial.toClass());
128         context.addType(line, thisType);
129     }
130
131     /**
132      * Pre-analyze the members of this declaration in the parent context.
133      * Pre-analysis extends to the member headers (including method headers) but
134      * not into the bodies.
135      */

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

136     * @param context
137     *         the parent (compilation unit) context.
138     */
139
140     public void preAnalyze(Context context) {
141         // Construct a class context
142         this.context = new ClassContext(this, context);
143
144         // Resolve superclass
145         superType = superType.resolve(this.context);
146
147         // Creating a partial class in memory can result in a
148         // java.lang.VerifyError if the semantics below are
149         // violated, so we can't defer these checks to analyze()
150         thisType.checkAccess(line, superType);
151         if (superType.isFinal()) {
152             JAST.compilationUnit.reportSemanticError(line,
153                 "Cannot extend a final type: %s", superType.toString());
154         }
155
156         // Create the (partial) class
157         CLEmitter partial = new CLEmitter(false);
158
159         // Add the class header to the partial class
160         String qualifiedName = JAST.compilationUnit.packageName() == "" ? name
161             : JAST.compilationUnit.packageName() + "/" + name;
162         partial.addClass(mods, qualifiedName, superType.jvmName(), null, false);
163
164         // Pre-analyze the members and add them to the partial
165         // class
166         for (JMember member : classBlock) {
167             member.preAnalyze(this.context, partial);
168             if (member instanceof JConstructorDeclaration
169                 && ((JConstructorDeclaration) member).params.size() == 0) {
170                 hasExplicitConstructor = true;
171             }
172         }
173
174         // Add the implicit empty constructor
175         if (!hasExplicitConstructor) {
176             codegenPartialImplicitConstructor(partial);
177         }
178
179         // Get the Class rep for the (partial) class and make it
180         // the
181         // representation for this type
182         Type id = this.context.lookupType(name);
183         if (id != null && !JAST.compilationUnit.errorHasOccurred()) {
184             id.setClassRep(partial.toClass());
185         }
186     }
187
188     /**
189     * Perform semantic analysis on the class and all of its members within the
190     * given context. Analysis includes field initializations and the method
191     * bodies.
192     *
193     * @param context
194     *         the parent (compilation unit) context. Ignored here.
195     * @return the analyzed (and possibly rewritten) AST subtree.
196     */
197
198     public JAST analyze(Context context) {
199         // Analyze all members
200         for (JMember member : classBlock) {
201             ((JAST) member).analyze(this.context);
202         }
203
204         // Copy declared fields for purposes of initialization.

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

205     for (JMember member : classBlock) {
206         if (member instanceof JFieldDeclaration) {
207             JFieldDeclaration fieldDecl = (JFieldDeclaration) member;
208             if (fieldDecl.mods().contains("static")) {
209                 staticFieldInitializations.add(fieldDecl);
210             } else {
211                 instanceFieldInitializations.add(fieldDecl);
212             }
213         }
214     }
215
216     // Finally, ensure that a non-abstract class has
217     // no abstract methods.
218     if (!thisType.isAbstract() && thisType.abstractMethods().size() > 0) {
219         String methods = "";
220         for (Method method : thisType.abstractMethods()) {
221             methods += "\n" + method;
222         }
223         JAST.compilationUnit.reportSemanticError(line,
224             "Class must be declared abstract since it defines "
225             + "the following abstract methods: %s", methods);
226     }
227     return this;
228 }
229
230 /**
231  * Generate code for the class declaration.
232  * @param output the code emitter (basically an abstraction for producing the
233  *               .class file).
234  */
235
236 public void codegen(CLEmitter output) {
237     // The class header
238     String qualifiedName = JAST.compilationUnit.packageName() == "" ? name
239         : JAST.compilationUnit.packageName() + "/" + name;
240     output.addClass(new QualifiedName(), superType.fullName(), null, false);
241
242     // The implicit empty constructor?
243     if (!hasExplicitConstructor) {
244         codegenImplicitConstructor(output);
245     }
246
247     // The members
248     for (JMember member : classBlock) {
249         ((JAST) member).codegen(output);
250     }
251
252     // Generate a class initialization method?
253     if (staticFieldInitializations.size() > 0) {
254         codegenClassInit(output);
255     }
256 }
257
258 /**
259  * @inheritDoc
260  */
261
262 public void writeToStdOut(PrettyPrinter p) {
263     p.printf("<JClassDeclaration line=\"%d\" name=\"%s\" "
264         + "super=\"%s\">\n", line(), name, superType.toString());
265     p.indentRight();
266     if (context != null) {
267         context.writeToStdOut(p);
268     }
269     if (mods != null) {
270         p.println("<Modifiers>");
271     }
272 }
273

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

274         p.indentRight();
275         for (String mod : mods) {
276             p.printf("<Modifier name=\"%s\"/>\n", mod);
277         }
278         p.indentLeft();
279         p.println("</Modifiers>");
280     }
281     if (classBlock != null) {
282         p.println("<ClassBlock>");
283         for (JMember member : classBlock) {
284             ((JAST) member).writeToStdOut(p);
285         }
286         p.println("</ClassBlock>");
287     }
288     p.indentLeft();
289     p.println("</JClassDeclaration>");
290 }
291
292 /**
293  * Generate code for an implicit empty constructor. (Necessary only if there
294  * is not already an explicit one.)
295  *
296  * @param partial
297  *     the code emitter (basically an abstraction for producing a
298  *     Java class).
299  */
300
301 private void codegenPartialImplicitConstructor(CLEmitter partial) {
302     // Invoke super constructor
303     ArrayList<String> mods = new ArrayList<String>();
304     mods.add("public");
305     partial.addMethod(mods, "<init>", "()"V", null, false);
306     partial.addNoArgInstruction(ALOAD_0);
307     partial.addMemberAccessInstruction(INVOKEVIRTUAL, superType.jvmName(),
308         "<init>", "()"V");
309
310     // Return
311     partial.addNoArgInstruction(RETURN);
312 }
313
314 /**
315  * Generate code for an implicit empty constructor. (Necessary only if there
316  * is not already an explicit one.
317  *
318  * @param output
319  *     the code emitter (basically an abstraction for producing the
320  *     .class file).
321  */
322
323 private void codegenImplicitConstructor(CLEmitter output) {
324     // Invoke super constructor
325     ArrayList<String> mods = new ArrayList<String>();
326     mods.add("public");
327     output.addMethod(mods, "<init>", "()"V", null, false);
328     output.addNoArgInstruction(ALOAD_0);
329     output.addMemberAccessInstruction(INVOKEVIRTUAL, superType.jvmName(),
330         "<init>", "()"V");
331
332     // If there are instance field initializations, generate
333     // code for them
334     for (JFieldDeclaration instanceField : instanceFieldInitializations) {
335         instanceField.codegenInitializations(output);
336     }
337
338     // Return
339     output.addNoArgInstruction(RETURN);
340 }
341
342 /**

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

343 * Generate code for class initialization, in j-- this means static field
344 * initializations.
345 *
346 * @param output
347 *         the code emitter (basically an abstraction for producing the
348 *         .class file).
349 */
350
351 private void codegenClassInit(CLEmitter output) {
352     ArrayList<String> mods = new ArrayList<String>();
353     mods.add("public");
354     mods.add("static");
355     output.addMethod(mods, "<clinit>", "()V", null, false);
356
357     // If there are instance initializations, generate code
358     // for them
359     for (JFieldDeclaration staticField : staticFieldInitializations) {
360         staticField.codegenInitializations(output);
361     }
362
363     // Return
364     output.addNoArgInstruction(RETURN);
365 }
366
367 }
368

```

# Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder