```
1   // Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3   package jminusminus;
4
5   import java.util.ArrayList;
6
7   /**
8    * The AST node for a field declaration.
9    */
10
11  class JFieldDeclaration extends JAST implements JMember {
12
13      /** Field modifiers. */
14      private ArrayList<String> mods;
15
16      /** Variable declarators. */
17      private ArrayList<JVariableDeclarator> decls;
18
19      /** Variable initializations. */
20      private ArrayList<JStatement> initializations;
21
22      /**
23       * Construct an AST node for a field declaration given the line number,
24       * modifiers, and the variable declarators.
25       *
26       * @param line
27       *             line in which the variable declaration occurs in the source
28       *             file.
29       * @param mods
30       *             field modifiers.
31       * @param decls
32       *             variable declarators.
33       */
34
35      public JFieldDeclaration(int line, ArrayList<String> mods,
36              ArrayList<JVariableDeclarator> decls) {
37          super(line);
38          this.mods = mods;
39          this.decls = decls;
40          initializations = new ArrayList<JStatement>();
41      }
42
43      /**
44       * Return the list of modifiers.
45       *
46       * @return list of modifiers.
47       */
48
49      public ArrayList<String> mods() {
50          return mods;
51      }
52
53      /**
54       * Declare fields in the parent's (partial) class.
55       *
56       * @param context
57       *             the parent (class) context.
58       * @param partial
59       *             the code emitter (basically an abstraction for producing the
60       *             partial class).
61       */
62
63      public void preAnalyze(Context context, CLEmitter partial) {
64          // Fields may not be declared abstract.
65          if (mods.contains("abstract")) {
66              JAST.compilationUnit.reportSemanticError(line(),
```

```
                    "Field cannot be declared abstract");
        }

        for (JVariableDeclarator decl : decls) {
            // Add field to (partial) class
            decl.setType(decl.type().resolve(context));
            partial.addField(mods, decl.name(), decl.type().toDescriptor(),
                    false);
        }
    }

    /**
     * Analysis of field declaration involves rewriting initializations (if any)
     * as assignment statements.
     *
     * @param context
     *            context in which names are resolved.
     * @return the analyzed JFieldDeclaration subtree.
     */

    public JFieldDeclaration analyze(Context context) {
        for (JVariableDeclarator decl : decls) {
            // All initializations must be turned into assignment
            // statements and analyzed
            if (decl.initializer() != null) {
                JAssignOp assignOp = new JAssignOp(decl.line(), new JVariable(
                        decl.line(), decl.name()), decl.initializer());
                assignOp.isStatementExpression = true;
                initializations.add(new JStatementExpression(decl.line(),
                        assignOp).analyze(context));
            }
        }
        return this;
    }

    /**
     * Generate code for any field initializations (now rewritten as assignment
     * statements).
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    public void codegenInitializations(CLEmitter output) {
        for (JStatement initialization : initializations) {
            initialization.codegen(output);
        }
    }

    /**
     * Code generation for field declaration involves generate field the header.
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    public void codegen(CLEmitter output) {
        for (JVariableDeclarator decl : decls) {
            // Add field to class
            output.addField(mods, decl.name(), decl.type().toDescriptor(),
                    false);
        }
    }

    /**
     * @inheritDoc
     */
```

```
136
137     public void writeToStdOut(PrettyPrinter p) {
138         p.printf("<JFieldDeclaration line=\"%d\"/>\n", line());
139         p.indentRight();
140         if (mods != null) {
141             p.println("<Modifiers>");
142             p.indentRight();
143             for (String mod : mods) {
144                 p.printf("<Modifier name=\"%s\"/>\n", mod);
145             }
146             p.indentLeft();
147             p.println("</Modifiers>");
148         }
149         if (decls != null) {
150             p.println("<VariableDeclarators>");
151             for (JVariableDeclarator decl : decls) {
152                 p.indentRight();
153                 decl.writeToStdOut(p);
154                 p.indentLeft();
155             }
156             p.println("<VariableDeclarators>");
157         }
158         p.indentLeft();
159         p.println("</JFieldDeclaration>");
160     }
161
162 }
163
```