```java
import components.map.Map;

/**
 * {@code Program} represented the obvious way with implementations of primary
 * methods.
 *
 * @convention [$this.name is an IDENTIFIER] and [$this.context is a CONTEXT]
 *             and [$this.body is a BLOCK statement]
 * @correspondence this = ($this.name, $this.context, $this.body)
 *
 * @author Shyam Sai Bethina and Yihone Chu
 *
 */
public class Program2 extends ProgramSecondary {

    /*
     * Private members
    ------------------------------------------------------------
     */

    /**
     * The program name.
     */
    private String name;

    /**
     * The program context.
     */
    private Map<String, Statement> context;

    /**
     * The program body.
     */
    private Statement body;

    /**
     * Reports whether all the names of instructions in {@code c} are valid
     * IDENTIFIERs.
     *
     * @param c
```

```
47      *            the context to check
48      * @return true if all instruction names are identifiers;
   false otherwise
49      * @ensures <pre>
50      * allIdentifiers =
51      *   [all the names of instructions in c are valid
   IDENTIFIERs]
52      * </pre>
53      */
54     private static boolean allIdentifiers(Map<String,
   Statement> c) {
55         for (Map.Pair<String, Statement> pair : c) {
56             if (!Tokenizer.isIdentifier(pair.key())) {
57                 return false;
58             }
59         }
60         return true;
61     }
62 
63     /**
64      * Reports whether no instruction name in {@code c} is the
   name of a
65      * primitive instruction.
66      *
67      * @param c
68      *            the context to check
69      * @return true if no instruction name is the name of a
   primitive
70      *         instruction; false otherwise
71      * @ensures <pre>
72      * noPrimitiveInstructions =
73      *   [no instruction name in c is the name of a primitive
   instruction]
74      * </pre>
75      */
76     private static boolean noPrimitiveInstructions(Map<String,
   Statement> c) {
77         return !c.hasKey("move") && !c.hasKey("turnleft")
78                 && !c.hasKey("turnright") && !
   c.hasKey("infect")
79                 && !c.hasKey("skip");
80     }
81 
82     /**
```

```
 83      * Reports whether all the bodies of instructions in {@code
    c} are BLOCK
 84      * statements.
 85      *
 86      * @param c
 87      *            the context to check
 88      * @return true if all instruction bodies are BLOCK
    statements; false
 89      *         otherwise
 90      * @ensures <pre>
 91      * allBlocks =
 92      *   [all the bodies of instructions in c are BLOCK
    statements]
 93      * </pre>
 94      */
 95     private static boolean allBlocks(Map<String, Statement> c)
    {
 96         for (Map.Pair<String, Statement> pair : c) {
 97             if (pair.value().kind() != Kind.BLOCK) {
 98                 return false;
 99             }
100         }
101         return true;
102     }
103
104     /**
105      * Creator of initial representation.
106      */
107     private void createNewRep() {
108
109         /*
110          * Creates the new empty representation based on the
    slides in class.
111          */
112         this.name = "Unnamed";
113         this.context = new Map1L<String, Statement>();
114         this.body = new Statement1();
115
116         // Make sure to use Statement1 from the library
117         // Use Map1L for the context if you want the asserts
    below to match
118
119     }
120
```

```
121     /*
122      * Constructors
    ------------------------------------------------------------
123      */
124
125     /**
126      * No-argument constructor.
127      */
128     public Program2() {
129         /*
130          * Creates new representation by
    calling .createNewRep()
131          */
132         this.createNewRep();
133     }
134
135     /*
136      * Standard methods
    ----------------------------------------------------------
137      */
138
139     @Override
140     public final Program newInstance() {
141         try {
142             return
    this.getClass().getConstructor().newInstance();
143         } catch (ReflectiveOperationException e) {
144             throw new AssertionError(
145                     "Cannot construct object of type " +
    this.getClass());
146         }
147     }
148
149     @Override
150     public final void clear() {
151         this.createNewRep();
152     }
153
154     @Override
155     public final void transferFrom(Program source) {
156         assert source != null : "Violation of: source is not
    null";
157         assert source != this : "Violation of: source is not
    this";
```

```
158         assert source instanceof Program2 : ""
159                 + "Violation of: source is of dynamic type
    Program2";
160         /*
161          * This cast cannot fail since the assert above would
    have stopped
162          * execution in that case: source must be of dynamic
    type Program2.
163          */
164         Program2 localSource = (Program2) source;
165         this.name = localSource.name;
166         this.context = localSource.context;
167         this.body = localSource.body;
168         localSource.createNewRep();
169     }
170 
171     /*
172      * Kernel methods
    ------------------------------------------------------------
173      */
174 
175     @Override
176     public final void setName(String n) {
177         assert n != null : "Violation of: n is not null";
178         assert Tokenizer.isIdentifier(n) : ""
179                 + "Violation of: n is a valid IDENTIFIER";
180 
181         /*
182          * Sets the name of this to the string parameter.
183          */
184         this.name = n;
185 
186     }
187 
188     @Override
189     public final String name() {
190 
191         /*
192          * Returns this' name
193          */
194         return this.name;
195     }
196 
197     @Override
```

```
198     public final Map<String, Statement> newContext() {
199 
200         /*
201          * Returns an empty context with the same dynamic type
    as this.
202          */
203         return this.context.newInstance();
204     }
205 
206     @Override
207     public final void swapContext(Map<String, Statement> c) {
208         assert c != null : "Violation of: c is not null";
209         assert c instanceof Map1L<?, ?> : "Violation of: c is a
    Map1L<?, ?>";
210         assert allIdentifiers(
211                 c) : "Violation of: names in c are valid
    IDENTIFIERs";
212         assert noPrimitiveInstructions(c) : ""
213                 + "Violation of: names in c do not match the
    names"
214                 + " of primitive instructions in the BL
    language";
215         assert allBlocks(c) : "Violation of: bodies in c"
216                 + " are all BLOCK statements";
217 
218         /*
219          * Creates an empty temp variable with the same dynamic
    type as this.
220          * Then uses .transferFrom to swap the context of this
    and parameter c.
221          */
222         Map<String, Statement> temp =
    this.context.newInstance();
223         temp.transferFrom(this.context);
224         this.context.transferFrom(c);
225         c.transferFrom(temp);
226     }
227 
228     @Override
229     public final Statement newBody() {
230 
231         /*
232          * Returns an empty body with the same dynamic type as
    this.
```

```
233          */
234         return this.body.newInstance();
235     }
236 
237     @Override
238     public final void swapBody(Statement b) {
239         assert b != null : "Violation of: b is not null";
240         assert b instanceof Statement1 : "Violation of: b is a
    Statement1";
241         assert b.kind() == Kind.BLOCK : "Violation of: b is a
    BLOCK statement";
242 
243         /*
244          * Creates an empty temp variable with the same dynamic
    type as this.
245          * Then uses .transferFrom to swap the body of this and
    parameter b.
246          */
247         Statement temp = this.body.newInstance();
248         temp.transferFrom(this.body);
249         this.body.transferFrom(b);
250         b.transferFrom(temp);
251     }
252 
253 }
254 
```