

```

1 import components.simplereader.SimpleReader;
2
3 /**
4  * This program calculates the value of an expression consisting of numbers,
5  * arithmetic operators, and parentheses.
6  *
7  * @author Put your name here
8  */
9
10 public final class ExpressionEvaluator {
11
12     /**
13      * Base used in number representation.
14      */
15     private static final int RADIX = 10;
16
17     private static final String[] terms = { "0", "1", "2", "3", "4", "5", "6",
18         "7", "8", "9", "*", "/", "(", ")" };
19
20     private static final String[] factors = { "0", "1", "2", "3", "4", "5", "6",
21         "7", "8", "9", "(", ")" };
22
23     private static final String[] digits = { "0", "1", "2", "3", "4", "5", "6",
24         "7", "8", "9" };
25
26     public static SimpleWriter out = new SimpleWriter1L();
27
28     /**
29      * Private constructor so this utility class cannot be instantiated.
30      */
31     private ExpressionEvaluator() {
32
33     }
34
35     /**
36      * Evaluates a digit and returns its value.
37      *
38      * @param source
39      *     the {@code StringBuilder} that starts with a digit
40      * @return value of the digit
41      * @updates source
42      * @requires 1 < |source| and [the first character of source is a digit]
43      * @ensures <pre>
44      *     valueOfDigit = [value of the digit at the start of #source] and
45      *     #source = [digit string at start of #source] * source
46      * </pre>
47      */
48     private static int valueOfDigit(StringBuilder source) {
49         assert source != null : "Violation of: source is not null";
50
51         return Integer.valueOf(source.charAt(0));
52     }
53
54     /**
55      * Evaluates a digit sequence and returns its value.
56      *
57      * @param source
58      *     the {@code StringBuilder} that starts with a digit-seq string

```

```

61     * @return value of the digit sequence
62     * @updates source
63     * @requires <pre>
64     * [a digit-seq string is a proper prefix of source, which
65     * contains a character that is not a digit]
66     * </pre>
67     * @ensures <pre>
68     * valueOfDigitSeq =
69     * [value of longest digit-seq string at start of #source] and
70     * #source = [longest digit-seq string at start of #source] * source
71     * </pre>
72     */
73     private static int valueOfDigitSeq(StringBuilder source) {
74         assert source != null : "Violation of: source is not null";
75
76         int idx = 0;
77         StringBuilder value = new StringBuilder();
78         StringBuilder term = new StringBuilder();
79         StringBuilder next = new StringBuilder();
80
81         while (idx < source.length()) {
82             next.delete(0, next.length());
83             next.append(source.charAt(idx));
84             term.append(next);
85
86             for (String check : digits) {
87                 if (next.toString() == check) {
88                     value.append(Integer.toString(valueOfDigit(next)));
89                 } else {
90                     idx = source.length();
91                 }
92             }
93
94             idx++;
95         }
96
97         return Integer.valueOf(value.toString());
98     }
99
100    /**
101     * Evaluates a factor and returns its value.
102     *
103     * @param source
104     *     the {@code StringBuilder} that starts with a factor string
105     * @return value of the factor
106     * @updates source
107     * @requires <pre>
108     * [a factor string is a proper prefix of source, and the longest
109     * such, s, concatenated with the character following s, is not a prefix
110     * of any factor string]
111     * </pre>
112     * @ensures <pre>
113     * valueOfFactor =
114     * [value of longest factor string at start of #source] and
115     * #source = [longest factor string at start of #source] * source
116     * </pre>
117     */

```

```

118     private static int valueOfFactor(StringBuilder source) {
119         assert source != null : "Violation of: source is not null";
120
121         int value = 0;
122         StringBuilder digitSeq = new StringBuilder();
123         StringBuilder next = new StringBuilder();
124         StringBuilder source2 = new StringBuilder();
125
126         if (source.charAt(0) == '(') {
127             value = valueOfExpr(source);
128         } else {
129             value = valueOfDigitSeq(source);
130         }
131
132         return value;
133     }
134
135     /**
136     * Evaluates a term and returns its value.
137     *
138     * @param source
139     *     the {@code StringBuilder} that starts with a term string
140     * @return value of the term
141     * @updates source
142     * @requires <pre>
143     * [a term string is a proper prefix of source, and the longest
144     * such, s, concatenated with the character following s, is not a prefix
145     * of any term string]
146     * </pre>
147     * @ensures <pre>
148     * valueOfTerm =
149     * [value of longest term string at start of #source] and
150     * #source = [longest term string at start of #source] * source
151     * </pre>
152     */
153     private static int valueOfTerm(StringBuilder source) {
154         assert source != null : "Violation of: source is not null";
155
156         int value = 0;
157         int idx = 0;
158         StringBuilder factor = new StringBuilder();
159         StringBuilder next = new StringBuilder();
160         StringBuilder source2 = new StringBuilder();
161
162         while (idx < source.length()) {
163             next.delete(0, next.length());
164             next.append(source.charAt(idx));
165             factor.append(next);
166
167             for (String check : factors) {
168                 if (next.toString() == check) {
169                     // do nothing lol
170                 } else if (next.toString() == "*") {
171                     source2 = source;
172                     source2.delete(0, idx);
173                     value += valueOfFactor(factor) * valueOfTerm(source2);
174

```

```

175         } else if (next.toString() == "/") {
176             source2 = source;
177             source2.delete(0, idx);
178             value += valueOfFactor(factor) / valueOfTerm(source2);
179         } else {
180             idx = source.length();
181         }
182     }
183
184     idx++;
185 }
186
187 return value;
188 }
189
190 /**
191  * Evaluates an expression and returns its value.
192  *
193  * @param source
194  *     the {@code StringBuilder} that starts with an expr string
195  * @return value of the expression
196  * @updates source
197  * @requires <pre>
198  * [an expr string is a proper prefix of source, and the longest
199  * such, s, concatenated with the character following s, is not a prefix
200  * of any expr string]
201  * </pre>
202  * @ensures <pre>
203  * valueOfExpr =
204  * [value of longest expr string at start of #source] and
205  * #source = [longest expr string at start of #source] * source
206  * </pre>
207  */
208 public static int valueOfExpr(StringBuilder source) {
209     assert source != null : "Violation of: source is not null";
210
211     int value = 0;
212     int idx = 0;
213     StringBuilder term = new StringBuilder();
214     StringBuilder next = new StringBuilder();
215     StringBuilder source2 = new StringBuilder();
216
217     while (idx < source.length()) {
218         next.delete(0, next.length());
219         next.append(source.charAt(idx));
220
221         for (String check : terms) {
222             out.println(next.toString());
223             out.println(check);
224
225             if (next.toString() == check) {
226                 term.append(next);
227             }
228         }
229
230         if (next.toString() == "-") {
231             source2 = source;

```

```
232         source2.delete(0, idx);
233         value += valueOfTerm(term) - valueOfExpr(source2);
234         break;
235     } else if (next.toString() == "+") {
236         source2 = source;
237         source2.delete(0, idx);
238         value += valueOfTerm(term) + valueOfExpr(source2);
239         break;
240
241     }
242
243     idx++;
244 }
245
246
247
248
249 // This line added just to make the program compilable.
250 return value;
251
252 }
253
254 /**
255  * Main method.
256  *
257  * @param args
258  *     the command line arguments
259  */
260 public static void main(String[] args) {
261     SimpleReader in = new SimpleReader1L();
262     SimpleWriter out = new SimpleWriter1L();
263     out.print("Enter an expression followed by !: ");
264     String source = in.nextLine();
265     while (source.length() > 0) {
266         /*
267          * Parse and evaluate the expression after removing all white space
268          * (spaces and tabs) from the user input.
269          */
270         int value = valueOfExpr(
271             new StringBuilder(source.replaceAll("[ \\t]", "")));
272         out.println(
273             source.substring(0, source.length() - 1) + " = " + value);
274         out.print("Enter an expression followed by !: ");
275         source = in.nextLine();
276     }
277     in.close();
278     out.close();
279 }
280
281 }
282
```