

```

1 import java.util.Iterator;
2
3 /**
4  * {@code Stack} represented as a singly linked list, done "bare-handed", with
5  * implementations of primary methods.
6  */
7
8 * <p>
9 * Execution-time performance of all methods implemented in this class is O(1).
10 *
11 * @param <T>
12 *         type of Stack entries
13 * @pre
14 *   $this.length >= 0 and
15 *   if $this.length = 0 then
16 *     [$this.top is null]
17 *   else
18 *     [$this.top is not null] and
19 *     [$this.top points to the first node of a singly linked list
20 *     containing $this.length nodes] and
21 *     [next in the last node of that list is null]
22 * </pre>
23 * @correspondence this = [data in $this.length nodes starting at $this.top]
24 */
25
26 public class Stack2<T> extends StackSecondary<T> {
27
28     /*
29      * Private members -----
30      */
31
32     /**
33      * Node class for singly linked list nodes.
34      */
35     private final class Node {
36
37         /**
38          * Data in node.
39          */
40         private T data;
41
42         /**
43          * Next node in singly linked list, or null.
44          */
45         private Node next;
46     }
47
48     /**
49      * Top node of singly linked list.
50      */
51     private Node top;
52
53     /**
54      * Number of nodes in singly linked list, i.e., length = |this|.
55      */
56     private int length;
57
58     /**
59
60
61

```

```

62     * Checks that the part of the convention repeated below holds for the
63     * current representation.
64     *
65     * @return true if the convention holds (or if assertion checking is off);
66     *         otherwise reports a violated assertion
67     * @convention <pre>
68     * $this.length >= 0 and
69     * if $this.length == 0 then
70     *   [$this.top is null]
71     * else
72     *   [$this.top is not null] and
73     *   [$this.top points to the first node of a singly linked list
74     *   containing $this.length nodes] and
75     *   [next in the last node of that list is null]
76     * </pre>
77     */
78     private boolean conventionHolds() {
79         assert this.length >= 0 : "Violation of: $this.length >= 0";
80         if (this.length == 0) {
81             assert this.top == null : ""
82                 + "Violation of: if $this.length == 0 then $this.top is null";
83         } else {
84             assert this.top != null : ""
85                 + "Violation of: if $this.length > 0 then $this.top is not null";
86             int count = 0;
87             Node tmp = this.top;
88             while ((tmp != null) && (count < this.length)) {
89                 count++;
90                 tmp = tmp.next;
91             }
92             assert this.length == count : "Violation of: if $this.length > 0 then "
93                 + "[$this.top points to the first node of a singly "
94                 + "linked list containing $this.length nodes]";
95             assert tmp == null : "Violation of: if $this.length > 0 then "
96                 + "[$this.top points to the first node of a singly "
97                 + "linked list containing $this.length nodes] and "
98                 + "[next in the last node of that list is null]";
99         }
100         return true;
101     }
102
103     /**
104     * Creator of initial representation.
105     */
106     private void createNewRep() {
107
108         this.top = new Node();
109         this.length = 1;
110     }
111
112
113     /**
114     * Constructors -----
115     */
116
117     /**
118     * No-argument constructor.

```

```

119     */
120     public Stack2() {
121         this.createNewRep();
122         assert this.conventionHolds();
123     }
124
125     /*
126     * Standard methods -----
127     */
128
129     @SuppressWarnings("unchecked")
130     @Override
131     public final Stack<T> newInstance() {
132         try {
133             return this.getClass().getConstructor().newInstance();
134         } catch (ReflectiveOperationException e) {
135             throw new AssertionError(
136                 "Cannot construct object of type " + this.getClass());
137         }
138     }
139
140     @Override
141     public final void clear() {
142         this.createNewRep();
143         assert this.conventionHolds();
144     }
145
146     @Override
147     public final void transferFrom(Stack<T> source) {
148         assert source != null : "Violation of: source is not null";
149         assert source != this : "Violation of: source is not this";
150         assert source instanceof Stack2<?> : ""
151             + "Violation of: source is of dynamic type Stack2<?>";
152         /*
153         * This cast cannot fail since the assert above would have stopped
154         * execution in that case: source must be of dynamic type Stack2<?>, and
155         * the ? must be T or the call would not have compiled.
156         */
157         Stack2<T> localSource = (Stack2<T>) source;
158         this.top = localSource.top;
159         this.length = localSource.length;
160         localSource.createNewRep();
161         assert this.conventionHolds();
162         assert localSource.conventionHolds();
163     }
164
165     /*
166     * Kernel methods -----
167     */
168
169     @Override
170     public final void push(T x) {
171         assert x != null : "Violation of: x is not null";
172
173         // create new node
174         Node temp = new Node();
175         temp.data = x;

```

```
176
177     // rearrange nodes
178     temp.next = this.top;
179     this.top = temp;
180
181     this.length++;
182
183     assert this.conventionHolds();
184 }
185
186 @Override
187 public final T pop() {
188     assert this.length() > 0 : "Violation of: this /= <>";
189
190     T temp = this.top.data;
191
192     this.top = this.top.next;
193
194     this.length--;
195
196     assert this.conventionHolds();
197     // Fix this line to return the result after checking the convention.
198     return temp;
199 }
200
201 @Override
202 public final int length() {
203
204     assert this.conventionHolds();
205     // Fix this line to return the result after checking the convention.
206     return this.length;
207 }
208
209 @Override
210 public final Iterator<T> iterator() {
211     return new Stack2Iterator();
212 }
213
214 /**
215  * Implementation of {@code Iterator} interface for {@code Stack2}.
216  */
217 private final class Stack2Iterator implements Iterator<T> {
218
219     /**
220      * Current node in the linked list.
221      */
222     private Node current;
223
224     /**
225      * No-argument constructor.
226      */
227     private Stack2Iterator() {
228         this.current = Stack2.this.top;
229         assert Stack2.this.conventionHolds();
230     }
231
232     @Override
```

```
233     public boolean hasNext() {
234         assert Stack2.this.conventionHolds();
235         return this.current != null;
236     }
237
238     @Override
239     public T next() {
240         assert this.hasNext() : "Violation of: ~this.unseen /= <>";
241         if (!this.hasNext()) {
242             /*
243              * Exception is supposed to be thrown in this case, but with
244              * assertion-checking enabled it cannot happen because of assert
245              * above.
246              */
247             throw new NoSuchElementException();
248         }
249         T x = this.current.data;
250         this.current = this.current.next;
251         assert Stack2.this.conventionHolds();
252         return x;
253     }
254
255     @Override
256     public void remove() {
257         throw new UnsupportedOperationException(
258             "remove operation not supported");
259     }
260
261 }
262
263 }
264
```