

```
1 /**
2  * This class contains a collection of methods that help with
3  * testing. All methods
4  * here are static so there's no need to construct a Testing
5  * object. Just call them
6  * with the class name like so:
7  * <p></p>
8  * <code>Testing.assertEquals("test description", expected,
9  * actual)</code>
10 *
11 * @author Kristina Striegnitz, Aaron Cass, Chris Fernandes
12 * @version 5/28/18
13 */
14 public class Testing {
15
16     private static boolean VERBOSE = false;
17     private static int numTests;
18     private static int numFails;
19
20     /**
21      * Toggles between a lot of output and little output.
22      *
23      * @param verbose
24      *      If verbose is true, then complete
25      *      information is printed,
26      *      whether the tests passes or fails. If
27      *      verbose is false, only
28      *      failures are printed.
29      */
30     public static void setVerbose(boolean verbose)
31     {
32         VERBOSE = verbose;
33     }
34
35     /**
36      * Each of the assertEquals methods tests whether the
37      * actual
38      * result equals the expected result. If it does, then the
39      * test
40      * passes, otherwise it fails.
41      */
42 }
```

```

35      * The only difference between these methods is the types
      of the
36      * parameters.
37      *
38      * All take a String message and two values of some other
      type to
39      * compare:
40      *
41      * @param message
42      *           a message or description of the test
43      * @param expected
44      *           the correct, or expected, value
45      * @param actual
46      *           the actual value
47      */
48      public static void assertEquals(String message, boolean
      expected,
49                                     boolean actual)
50      {
51          printTestCaseInfo(message, "" + expected, "" + actual
      );
52          if (expected == actual) {
53              pass();
54          } else {
55              fail(message);
56          }
57      }
58
59      public static void assertEquals(String message, int
      expected, int actual)
60      {
61          printTestCaseInfo(message, "" + expected, "" + actual
      );
62          if (expected == actual) {
63              pass();
64          } else {
65              fail(message);
66          }
67      }
68
69      public static void assertEquals(String message, Object

```

```

69 expected,
70                                     Object actual)
71     {
72         String expectedString = "<<null>>";
73         String actualString = "<<null>>";
74         if (expected ≠ null) {
75             expectedString = expected.toString();
76         }
77         if (actual ≠ null) {
78             actualString = actual.toString();
79         }
80         printTestCaseInfo(message, expectedString,
actualString);
81
82         if (expected = null) {
83             if (actual = null) {
84                 pass();
85             } else {
86                 fail(message);
87             }
88         } else if (expected.equals(actual)) {
89             pass();
90         } else {
91             fail(message);
92         }
93     }
94
95     /**
96      * Asserts that a given boolean must be true. The test
fails if
97      * the boolean is not true.
98      *
99      * @param message The test message
100     * @param actual The boolean value asserted to be true.
101     */
102     public static void assertTrue(String message, boolean
actual)
103     {
104         assertEquals(message, true, actual);
105     }
106

```

```
107     /**
108      * Asserts that a given boolean must be false. The test
      fails if
109      * the boolean is not false (i.e. if it is true).
110      *
111      * @param message The test message
112      * @param actual The boolean value asserted to be false.
113      */
114     public static void assertFalse(String message, boolean
      actual)
115     {
116         assertEquals(message, false, actual);
117     }
118
119     private static void printTestCaseInfo(String message,
      String expected,
120                                         String actual)
121     {
122         if (VERBOSE) {
123             System.out.println(message + ":");
124             System.out.println("expected: " + expected);
125             System.out.println("actual:   " + actual);
126         }
127     }
128
129     private static void pass()
130     {
131         numTests++;
132
133         if (VERBOSE) {
134             System.out.println("--PASS--");
135             System.out.println();
136         }
137     }
138
139     private static void fail(String description)
140     {
141         numTests++;
142         numFails++;
143
144         if (!VERBOSE) {
```

```

145         System.out.print(description + " ");
146     }
147     System.out.println("--FAIL--");
148     System.out.println();
149 }
150
151 /**
152  * Prints a header for a section of tests.
153  *
154  * @param sectionTitle The header that should be printed.
155  */
156 public static void testSection(String sectionTitle)
157 {
158     if (VERBOSE) {
159         int dashCount = sectionTitle.length();
160         System.out.println(sectionTitle);
161         for (int i = 0; i < dashCount; i++) {
162             System.out.print("-");
163         }
164         System.out.println();
165         System.out.println();
166     }
167 }
168
169 /**
170  * Initializes the test suite. Should be called before
171  * running any
172  * tests, so that passes and fails are correctly tallied.
173  */
174 public static void startTests()
175 {
176     System.out.println("Starting Tests");
177     System.out.println();
178     numTests = 0;
179     numFails = 0;
180 }
181
182 /**
183  * Prints out summary data at end of tests. Should be
184  * called
185  * after all the tests have run.

```

```
184     */
185     public static void finishTests()
186     {
187         System.out.println("=====");
188         System.out.println("Tests Complete");
189         System.out.println("=====");
190         int numPasses = numTests - numFails;
191
192         System.out.print(numPasses + "/" + numTests + " PASS
193     ");
194         System.out.printf("(pass rate: %.1f%s)\n",
195                             100 * ((double) numPasses) /
196                             numTests,
197                             "%");
198         System.out.print(numFails + "/" + numTests + " FAIL "
199     );
200         System.out.printf("(fail rate: %.1f%s)\n",
201                             100 * ((double) numFails) /
202                             numTests,
203                             "%");
204     }
```

```
1  import java.util.ArrayList;
2
3  /**
4   * List Processor
5   *
6   * @author Chris Hegang Kim
7   * @note I affirm that I have carried out the attached
8   * academic endeavors with full academic honesty,
9   * in accordance with the Union College Honor Code and the
10  * course syllabus.
11  */
12 public class ListProcessor
13 {
14     /**
15      * Swaps elements i and j in the given list.
16      */
17     private void swap(ArrayList<String> aList, int i, int j)
18     {
19         String tmp = aList.get(i);
20         aList.set(i, aList.get(j));
21         aList.set(j, tmp);
22     }
23
24     /**
25      * Finds the minimum element of a list and returns it.
26      * Non-destructive (That means this method should not
27      * change aList.)
28      *
29      * @param aList the list in which to find the minimum
30      * element.
31      * @return the minimum element of the list.
32      */
33     public String getMin(ArrayList<String> aList) {
34         return getMin(aList, 0);
35     }
36
37     /**
38      * Finds the minimum element of a list from the
39      * startingIndex to the end and returns it.
```

```

37      *
38      * @param aList the list in which to find the minimum
    element.
39      * @param startingIndex the integer for the starting index
    .
40      * @return the minimum element of the list.
41      */
42      private String getMin(ArrayList<String> aList, int
    startingIndex) {
43          if (isEnd(aList, startingIndex)) {
44              return aList.get(startingIndex);
45          }
46
47          else {
48              String restOfTheList = getMin(aList, startingIndex
    + 1);
49
50              if (aList.get(startingIndex).compareTo(
    restOfTheList) > 0) {
51                  return restOfTheList;
52              }
53
54              else {
55                  return aList.get(startingIndex);
56              }
57          }
58      }
59
60
61      /**
62      * Finds the minimum element of a list and returns the
    index of that
63      * element. If there is more than one instance of the
    minimum, then
64      * the lowest index will be returned. Non-destructive.
65      *
66      * @param aList the list in which to find the minimum
    element.
67      * @return the index of the minimum element in the list.
68      */
69      public int getMinIndex(ArrayList<String> aList) {

```



```
70         return getMinIndex(aList, 0);
71     }
72
73
74     /**
75      * Finds the minimum element of a list from the
76      * startingIndex to the end
77      * and returns the index of that element.
78      * If there is more than one instance of the minimum,
79      * then the lowest index will be returned. Non-
80      * destructive.
81      *
82      * @param aList the list in which to find the minimum
83      * element.
84      * @param startingIndex the integer for the starting
85      * index.
86      * @return the index of the minimum element in the list.
87      */
88     private int getMinIndex(ArrayList<String> aList, int
89     startingIndex) {
90         if (isEnd(aList, startingIndex)) {
91             return startingIndex;
92         }
93
94         else {
95             if (aList.get(startingIndex).compareTo(getMin(
96             aList, startingIndex + 1)) > 0) {
97                 return getMinIndex(aList, startingIndex + 1);
98             }
99
100             else {
101                 return startingIndex;
102             }
103         }
104     }
105
106     /**
107      * Sorts a list in place. I.E. the list is modified so
108      * that it is in order.
109      */
```

```

104     * @param aList: the list to sort.
105     */
106     public void sort(ArrayList<String> aList) {
107         sort(aList, 0);
108     }
109
110
111     /**
112     * Sorts a list in place. I.E. the list is modified so
113     that it is in order.
114     *
115     * @param aList the list in which to find the minimum
116     element.
117     * @param startingIndex the integer for the starting
118     index.
119     */
120     private void sort(ArrayList<String> aList, int
121     startingIndex) {
122         if (!isEnd(aList, startingIndex)) {
123             int minIndex = getMinIndex(aList, startingIndex);
124             swap(aList, minIndex, startingIndex);
125             sort(aList, startingIndex + 1);
126         }
127     }
128
129     /**
130     * Checks whether the index is at the end.
131     *
132     * @param aList the list to check
133     * @param index the integer for the index
134     */
135     private boolean isEnd(ArrayList<String> aList, int index
136     ) {
137         if (index == (aList.size() - 1)) {
138             return true;
139         }
140
141         else {
142             return false;
143         }
144     }

```

```
140         }  
141     }  
142 }
```

```
1 import java.util.ArrayList;
2 import java.util.Arrays;
3
4 public class ListProcessorTester
5 {
6     public static void main(String [] args)
7     {
8         Testing.setVerbose(true);
9         Testing.startTests();
10        getMinTests();
11        getMinIndexTests();
12        sortTests();
13        Testing.finishTests();
14    }
15
16    /**
17     * turns an array of strings into an ArrayList
18     */
19    private static ArrayList<String> array2arraylist(String[]
strings){
20        return new ArrayList<String>(Arrays.asList(strings));
21    }
22
23    public static void getMinTests() {
24        Testing.testSection("Testing getMin");
25
26        ListProcessor lp = new ListProcessor();
27
28        String[] strings = {"b", "e", "a", "d", "g", "k", "c"
, "r", "t", "v", "a", "c", "b"};
29        ArrayList<String> originalList = array2arraylist(
strings);
30        ArrayList<String> copy = new ArrayList<String>(
originalList);
31        // makes a copy of originalList
32
33        String actual = lp.getMin(copy);
34        Testing.assertEquals("The minimum of a list of strings
is the first in alphabetical order",
35                               "a",
36                               actual);
```

```
37
38     Testing.assertEquals("getMin should not modify the
    list",
39         originalList,
40         copy);
41
42     actual = lp.getMin(array2arraylist(new String[]{"
aardvark", "lion", "zebra", "cougar", "cheetah"}));
43     Testing.assertEquals("boundary case: minimum in first
    position",
44         "aardvark",
45         actual);
46
47     actual = lp.getMin(array2arraylist(new String[]{"lion"
, "aardvark", "zebra", "cougar", "cheetah"}));
48     Testing.assertEquals("boundary case: minimum in second
    position",
49         "aardvark",
50         actual);
51
52     actual = lp.getMin(array2arraylist(new String[]{"lion"
, "zebra", "aardvark", "cougar", "cheetah"}));
53     Testing.assertEquals("boundary case: minimum in third
    position",
54         "aardvark",
55         actual);
56
57     actual = lp.getMin(array2arraylist(new String[]{"lion"
, "zebra", "cougar", "aardvark", "cheetah"}));
58     Testing.assertEquals("boundary case: minimum in fourth
    position",
59         "aardvark",
60         actual);
61
62     actual = lp.getMin(array2arraylist(new String[]{"bear"
, "lion", "zebra", "cougar", "antelope"}));
63     Testing.assertEquals("boundary case: minimum in last
    position",
64         "antelope",
65         actual);
66 }
```

```

67
68     public static void getMinIndexTests() {
69
70         Testing.testSection("Testing getMinIndex");
71
72         ListProcessor lp = new ListProcessor();
73         String[] strings = {"b", "e", "a", "d", "g", "k", "c",
74             , "r", "t", "v", "a", "c", "b"};
75         ArrayList<String> originalList = array2arraylist(
76             strings);
77         ArrayList<String> copy = new ArrayList<String>(
78             originalList);
79
80         Testing.assertEquals("getMinIndex should return the
81             index of the first occurrence of the min element",
82             2,
83             lp.getMinIndex(copy));
84
85         Testing.assertEquals("getMinIndex should not modify
86             the list",
87             originalList,
88             copy);
89
90         int actual = lp.getMinIndex(array2arraylist
91             (new String[]{"aardvark", "lion", "zebra", "
92             cougar", "cheetah"}));
93         Testing.assertEquals("boundary case: minimum in first
94             position",
95             0,
96             actual);
97
98         actual = lp.getMinIndex(array2arraylist(new String[]{
99             "lion", "aardvark", "zebra", "cougar", "cheetah"}));
100        Testing.assertEquals("boundary case: minimum in
101            second position",
102            1,
103            actual);
104
105        actual = lp.getMinIndex(array2arraylist(new String[]{
106            "lion", "zebra", "aardvark", "cougar", "cheetah"}));
107        Testing.assertEquals("boundary case: minimum in third

```

```

97     position",
98         2,
99         actual);
100
101     actual = lp.getMinIndex(array2arraylist(new String[]{
102         "lion", "zebra", "cougar", "aardvark", "cheetah"}));
103     Testing.assertEquals("boundary case: minimum in
104         fourth position",
105         3,
106         actual);
107
108     actual = lp.getMinIndex(array2arraylist
109         (new String[]{"bear", "lion", "zebra", "
110         cougar", "antelope"}));
111     Testing.assertEquals("boundary case: minimum in last
112         position",
113         4,
114         actual);
115
116     actual = lp.getMinIndex(array2arraylist
117         (new String[]{"antelope", "lion", "zebra", "
118         cougar", "antelope"}));
119     Testing.assertEquals("boundary case: minimum in first
120         and last position",
121         0,
122         actual);
123 }
124
125 public static void sortTests()
126 {
127     Testing.testSection("Testing sort");
128
129     ListProcessor lp = new ListProcessor();
130
131     String[] strings = {"b", "e", "a", "d", "g", "k", "c"
132 , "r", "t", "v", "a", "c", "b"};
133
134     ArrayList<String> myList = array2arraylist(strings);
135
136     lp.sort(myList);
137
138 }

```

```
131         String[] sortedStrings = {"a", "a", "b", "b", "c", "c",
    ", "d", "e", "g", "k", "r", "t", "v"};
132         ArrayList<String> sortedList = array2arraylist(
    sortedStrings);
133         Testing.assertEquals("sort puts list in alphabetic
    order",
134                                sortedList,
135                                myList);
136     }
137 }
```