

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

```

31 the actual
32     * result equals the expected result. If it does,
    then the test
33     * passes, otherwise it fails.
34     *
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,
70                                     Object expected,
71                                     Object actual)
72     {
73         String expectedString = "<<null>>";
74         String actualString = "<<null>>";
75         if (expected != null) {
76             expectedString = expected.toString();
77         }
78         if (actual != null) {
79             actualString = actual.toString();
80         }
81         printTestCaseInfo(message, expectedString,
82                             actualString);
83
84         if (expected == null) {
85             if (actual == null) {
86                 pass();
87             } else {
88                 fail(message);
89             }
90         } else if (expected.equals(actual)) {
91             pass();
92         } else {
93             fail(message);
94         }
95     }
96     /**
97      * Asserts that a given boolean must be true. The
98      * 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
155      * printed.
156      */
157     public static void testSection(String sectionTitle
158     )
159     {
160         if (VERBOSE) {
161             int dashCount = sectionTitle.length();
162             System.out.println(sectionTitle);
163             for (int i = 0; i < dashCount; i++) {
164                 System.out.print("-");
165             }
166             System.out.println();
167         }
168     }
169 }
```

```

163         }
164         System.out.println();
165         System.out.println();
166     }
167 }
168
169 /**
170  * Initializes the test suite. Should be called
    before running any
171  * tests, so that passes and fails are correctly
    tallied.
172  */
173 public static void startTests()
174 {
175     System.out.println("Starting Tests");
176     System.out.println();
177     numTests = 0;
178     numFails = 0;
179 }
180
181 /**
182  * Prints out summary data at end of tests.
    Should be called
183  * 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     System.out.printf("(pass rate: %.1f%s)\n",
194         100 * ((double) numPasses
    ) / numTests,
195         "%");

```

```
196
197         System.out.print(numFails + "/" + numTests +
    " FAIL ");
198         System.out.printf("(fail rate: %.1f%s)\n",
199                             100 * ((double) numFails) /
    numTests,
200                             "%");
201     }
202
203 }
204
```

```
1 import java.util.ArrayList;
2
3 public class ListProcessor
4 {
5     /**
6      * Swaps elements at indices i and j in the given
7      * list.
8      */
9     private void swap(ArrayList<String> aList, int i,
10     int j)
11     {
12         String tmp = aList.get(i);
13         aList.set(i, aList.get(j));
14         aList.set(j, tmp);
15     }
16
17     /**
18      * Finds the minimum element of a list and returns
19      * it.
20      * Non-destructive (That means this method should
21      * not change aList.)
22      *
23      * @param aList the list in which to find the
24      * minimum element.
25      * @return the minimum element of the list.
26      */
27     private String getMin(ArrayList<String> aList, int
28     startingIndex)
29     {
30         if (startingIndex == aList.size()-1) {
31             return aList.get(startingIndex);
32         }else if (aList.get(startingIndex).compareTo(
33         getMin(aList, startingIndex+1)) < 0) {
34             return aList.get(startingIndex);
35         }else {
36             return getMin(aList, startingIndex + 1);
37         }
38     }
39 }
```

```

32     }
33
34     /**
35      * Returns the minimum element in the list.
36      *
37      * @param aList the list to search
38      * @return the minimum element
39      */
40     public String getMin(ArrayList<String> aList){
41         return getMin(aList, 0);
42     }
43
44
45     /**
46      * Finds the minimum element of a list and returns
47      the index of that
48      * element. If there is more than one instance of
49      the minimum, then
50      * the lowest index will be returned. Non-
51      destructive.
52      *
53      * @param aList the list in which to find the
54      minimum element.
55      * @return the index of the minimum element in the
56      list.
57      */
58     private int getMinIndex(ArrayList<String> aList,
59 int startingIndex)
60     {
61         if (startingIndex == aList.size() - 1) {
62             return startingIndex;
63         }
64         if (aList.get(startingIndex).compareTo(aList.
65 get(getMinIndex(aList, startingIndex + 1))) <= 0) {
66             return startingIndex;
67         } else {
68             return getMinIndex(aList, startingIndex + 1
69 );
70 };

```

```

62         }
63     }
64
65     /**
66      * Returns the index of the minimum element in the
        list.
67      *
68      * @param aList the list to search
69      * @return the index of the first occurrence of the
        minimum element
70      */
71     public int getMinIndex(ArrayList<String> aList)
72     {
73         return getMinIndex(aList, 0);
74     }
75
76
77     /**
78      * Sorts a list in place. I.E. the list is modified
        so that it is in order.
79      *
80      * @param aList: the list to sort.
81      */
82     private void sort(ArrayList<String> aList, int
        startingIndex)
83     {
84         if (startingIndex >= aList.size() - 1) {
85             return;
86         }
87
88         swap(aList, getMinIndex(aList, startingIndex),
        startingIndex);
89
90         sort(aList, startingIndex + 1);
91     }
92
93     /**
94      * Sorts the list in alphabetical order using

```

```
94 selection sort.
95     *
96     * @param aList the list to sort (modified in
    place)
97     */
98     public void sort(ArrayList<String> aList)
99     {
100         sort(aList, 0);
101     }
102 }
103
104
105
```

```
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(
20    String[] strings){
21        return new ArrayList<String>(Arrays.asList(
22    strings));
23    }
24
25    public static void getMinTests() {
26        Testing.testSection("Testing getMin");
27
28        ListProcessor lp = new ListProcessor();
29
30        String[] strings = {"b", "e", "a", "d", "g", "k",
31    ", "c", "r", "t", "v", "a", "c", "b"};
32        ArrayList<String> originalList =
33    array2arraylist(strings);
34        ArrayList<String> copy = new ArrayList<String>(
35    originalList);
36        // makes a copy of originalList
37
38        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
[] {"bear", "lion", "zebra", "cougar", "antelope"}));
48         Testing.assertEquals("boundary case: minimum in
last position",
49             "antelope",
50             actual);
51     }
52
53
54     public static void getMinIndexTests() {
55
56         Testing.testSection("Testing getMinIndex");
57
58         ListProcessor lp = new ListProcessor();
59         String[] strings = {"b", "e", "a", "d", "g", "k",
" ", "c", "r", "t", "v", "a", "c", "b"};
60         ArrayList<String> originalList =
array2arraylist(strings);
61         ArrayList<String> copy = new ArrayList<String>(
originalList);
62
```

```

63         Testing.assertEquals("getMinIndex should
        return the index of the first occurrence of the min
        element",
64                                 2,
65                                 lp.getMinIndex(copy));
66
67         Testing.assertEquals("getMinIndex should not
        modify the list",
68                                 originalList,
69                                 copy);
70
71         int actual = lp.getMinIndex(array2arraylist
72                                     (new String[]{"aardvark", "lion", "
        zebra", "cougar", "cheetah"}));
73         Testing.assertEquals("boundary case: minimum
        in first position",
74                                 0,
75                                 actual);
76
77         actual = lp.getMinIndex(array2arraylist
78                                     (new String[]{"bear", "lion", "zebra"
        , "cougar", "antelope"}));
79         Testing.assertEquals("boundary case: minimum
        in last position",
80                                 4,
81                                 actual);
82
83     }
84
85     public static void sortTests()
86     {
87         Testing.testSection("Testing sort");
88
89         ListProcessor lp = new ListProcessor();
90
91         String[] strings = {"b", "e", "a", "d", "g", "
        k", "c", "r", "t", "v", "a", "c", "b"};
92

```

```
93         ArrayList<String> myList = array2arraylist(
           strings);
94
95         lp.sort(myList);
96
97         String[] sortedStrings = {"a", "a", "b", "b",
           "c", "c", "d", "e", "g", "k", "r", "t", "v"};
98         ArrayList<String> sortedList = array2arraylist
           (sortedStrings);
99         Testing.assertEquals("sort puts list in
           alphabetic order",
100                                sortedList,
101                                myList);
102     }
103
104 }
105
```