```java
/**
 * This class contains a collection of methods that help with testing.  All methods
 * here are static so there's no need to construct a Testing object.  Just call them
 * with the class name like so:
 * <p></p>
 * <code>Testing.assertEquals("test description", expected, actual)</code>
 *
 * @author Kristina Striegnitz, Aaron Cass, Chris Fernandes
 * @version 5/28/18
 */
public class Testing {

    private static boolean VERBOSE = false;
    private static int numTests;
    private static int numFails;

    /**
     * Toggles between a lot of output and little output.
     *
     * @param verbose
     *            If verbose is true, then complete information is printed,
     *            whether the tests passes or fails. If verbose is false, only
     *            failures are printed.
     */
    public static void setVerbose(boolean verbose)
    {
        VERBOSE = verbose;
    }

    /**
     * Each of the assertEquals methods tests whether the actual
     * 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
    , Object 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 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 
```

```java
import java.util.ArrayList;

/**
 * @author Neil Daterao
 * @note I affirm that I have carried out the attached academic endeavors with full academic honesty, in
 * accordance with the Union College Honor Code and the course syllabus.
 */
public class ListProcessor
{
    /**
     * Swaps elements i and j in the given list.
     */
    private void swap(ArrayList<String> aList, int i, int j)
    {
        String tmp = aList.get(i);
        aList.set(i, aList.get(j));
        aList.set(j, tmp);
    }

    /**
     * Finds the minimum element of a list and returns it.
     * Non-destructive (That means this method should not change aList.)
     *
     * @param aList the list in which to find the minimum element.
     * @return the minimum element of the list.
     */
    public String getMin(ArrayList<String> aList)
    {
        return getMin(aList, 0 );
    }

    /**
     * Finds the minimum element of a list and returns it.
```

```
34      * Non-destructive (That means this method
   should not change aList.)
35      * Helper method for public getMin function
36      *
37      * @param aList the list in which to find the
   minimum element.
38      * @param startingIndex starting index to search
    from
39      * @return
40      */
41     private String getMin(ArrayList<String> aList,
   int startingIndex) {
42         String currentMin = aList.get(startingIndex
   );
43         String restOfListMin = new String();
44
45        if (aList.size() - startingIndex == 1) {
46             return aList.get(startingIndex);
47        }
48        else {
49            restOfListMin = getMin(aList,
   startingIndex + 1);
50            return (currentMin.compareTo(
   restOfListMin) < 0) ? currentMin : restOfListMin;
51        }
52
53     }
54
55
56     /**
57      * Finds the minimum element of a list and
   returns the index of that
58      * element. If there is more than one instance
   of the minimum, then
59      * the lowest index will be returned.  Non-
   destructive.
60      *
61      * @param aList the list in which to find the
   minimum element.
62      * @return the index of the minimum element in
   the list.
```

```
63      */
64     public int getMinIndex(ArrayList<String> aList)
65     {
66         return getMinIndex(aList, 0, 1);
67     }
68 
69     /**
70      * Finds the minimum element of a list and
   returns the index of that
71      * element. If there is more than one instance
   of the minimum, then
72      * the lowest index will be returned.  Non-
   destructive.
73      * @param aList the list in which to find the
   minimum element.
74      * @param minIndex minimum index. to start,
   this is 0.
75      * @param currentIndex current index. to start
   , this is 1.
76      * @return
77      */
78     private int getMinIndex(ArrayList<String> aList
   , int minIndex, int currentIndex) {
79         if (currentIndex >= aList.size()) {
80             return minIndex;
81         }
82 
83         else if (aList.get(currentIndex).compareTo(
   aList.get(minIndex)) < 0) {
84             minIndex = currentIndex;
85         }
86 
87         return getMinIndex(aList, minIndex,
   currentIndex + 1);
88 
89     }
90 
91     /**
92      * Sorts a list in place. I.E. the list is
   modified so that it is in order.
93      *
```

```
 94      * @param aList: the list to sort.
 95      */
 96     public void sort(ArrayList<String> aList)
 97     {
 98         sort(aList, 0);
 99     }
100 
101     /**
102      * Selection sort algorithm for helper function
     of the public version of sort.
103      *
104      * @param aList the list to sort.
105      * @param startingIndex Starting index of list
106      */
107     private void sort(ArrayList<String> aList, int
    startingIndex) {
108 
109         if (startingIndex == aList.size()) {
110             return;
111         }
112         else {
113             int minIndex = getMinIndex(aList,
    startingIndex, startingIndex+1);
114             swap(aList, minIndex, startingIndex);
115             sort(aList, startingIndex + 1);
116         }
117 
118     }
119 }
120 
121 
122 
123 
```

```java
import java.util.ArrayList;
import java.util.Arrays;

public class ListProcessorTester
{
    public static void main(String [] args)
    {
        Testing.setVerbose(true);
        Testing.startTests();
        getMinTests();
        getMinIndexTests();
        sortTests();
        Testing.finishTests();
    }

    /**
     * turns an array of strings into an ArrayList
     */
    private static ArrayList<String> array2arraylist(String[] strings){
        return new ArrayList<String>(Arrays.asList(strings));
    }

    public static void getMinTests() {
        Testing.testSection("Testing getMin");

        ListProcessor lp = new ListProcessor();

        String[] strings = {"b", "e", "a", "d", "g", "k", "c", "r", "t", "v", "a", "c", "b"};
        ArrayList<String> originalList = array2arraylist(strings);
        ArrayList<String> copy = new ArrayList<String>(originalList);
        // makes a copy of originalList

        String actual = lp.getMin(copy);
        Testing.assertEquals("The minimum of a list of strings is the first in alphabetical order",
                "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
```