```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
     * passes, otherwise it fails.
     *
     * The only difference between these methods is the types of the
     * parameters.
     *
     * All take a String message and two values of some other type to
     * compare:
     *
     * @param message
     *            a message or description of the test
     * @param expected
     *            the correct, or expected, value
     * @param actual
     *            the actual value
     */
    public static void assertEquals(String message, boolean expected,
                                    boolean actual)
    {
        printTestCaseInfo(message, "" + expected, "" + actual);
        if (expected == actual) {
            pass();
        } else {
            fail(message);
        }
    }

    public static void assertEquals(String message, int expected, int actual)
    {
        printTestCaseInfo(message, "" + expected, "" + actual);
        if (expected == actual) {
            pass();
        } else {
            fail(message);
        }
    }

    public static void assertEquals(String message, Object
expected,
                                    Object actual)
    {
        String expectedString = "<<null>>";
        String actualString = "<<null>>";
        if (expected ≠ null) {
            expectedString = expected.toString();
        }
        if (actual ≠ null) {
            actualString = actual.toString();
        }
        printTestCaseInfo(message, expectedString,
actualString);

        if (expected == null) {
            if (actual == null) {
                pass();
            } else {
                fail(message);
            }
        } else if (expected.equals(actual)) {
            pass();
        } else {
            fail(message);
        }
    }

    /**
     * Asserts that a given boolean must be true.  The test
fails if
     * the boolean is not true.
     *
     * @param message The test message
     * @param actual The boolean value asserted to be true.
     */
    public static void assertTrue(String message, boolean
actual)
    {
        assertEquals(message, true, actual);
    }

```

```java
    /**
     * Asserts that a given boolean must be false. The test fails if
     * the boolean is not false (i.e. if it is true).
     *
     * @param message The test message
     * @param actual The boolean value asserted to be false.
     */
    public static void assertFalse(String message, boolean actual)
    {
        assertEquals(message, false, actual);
    }

    private static void printTestCaseInfo(String message, String expected,
                                          String actual)
    {
        if (VERBOSE) {
            System.out.println(message + ":");
            System.out.println("expected: " + expected);
            System.out.println("actual:   " + actual);
        }
    }

    private static void pass()
    {
        numTests++;

        if (VERBOSE) {
            System.out.println("--PASS--");
            System.out.println();
        }
    }

    private static void fail(String description)
    {
        numTests++;
        numFails++;

        if (!VERBOSE) {
            System.out.print(description + "  ");
        }
        System.out.println("--FAIL--");
        System.out.println();
    }

    /**
     * Prints a header for a section of tests.
     *
     * @param sectionTitle The header that should be printed.
     */
    public static void testSection(String sectionTitle)
    {
        if (VERBOSE) {
            int dashCount = sectionTitle.length();
            System.out.println(sectionTitle);
            for (int i = 0; i < dashCount; i++) {
                System.out.print("-");
            }
            System.out.println();
            System.out.println();
        }
    }

    /**
     * Initializes the test suite. Should be called before running any
     * tests, so that passes and fails are correctly tallied.
     */
    public static void startTests()
    {
        System.out.println("Starting Tests");
        System.out.println();
        numTests = 0;
        numFails = 0;
    }

    /**
     * Prints out summary data at end of tests.  Should be called
     * after all the tests have run.
     */
    public static void finishTests()
    {
        System.out.println("==============");
        System.out.println("Tests Complete");
        System.out.println("==============");
        int numPasses = numTests - numFails;

        System.out.print(numPasses + "/" + numTests + " PASS ");
        System.out.printf("(pass rate: %.1f%s)\n",
                          100 * ((double) numPasses) / numTests,
                          "%");

        System.out.print(numFails + "/" + numTests + " FAIL ");
        System.out.printf("(fail rate: %.1f%s)\n",
                          100 * ((double) numFails) / numTests,
                          "%");
    }

}
```

```java
import java.util.ArrayList;

/**
 * List Processor
 *
 * @author Chris Hegang Kim
 * @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 from the startingIndex to the end and returns it.
     *
     * @param aList the list in which to find the minimum
element.
     * @param startingIndex the integer for the starting index
.
     * @return the minimum element of the list.
     */
    private String getMin(ArrayList<String> aList, int
startingIndex) {
        if (isEnd(aList, startingIndex)) {
            return aList.get(startingIndex);
        }

        else {
            String restOfTheList = getMin(aList, startingIndex
 + 1);

            if (aList.get(startingIndex).compareTo(
restOfTheList) > 0) {
                return restOfTheList;
            }

            else {
                return aList.get(startingIndex);
            }
        }
    }


    /**
     * Finds the minimum element of a list and returns the
index of that
     * element. If there is more than one instance of the
minimum, then
     * the lowest index will be returned.  Non-destructive.
     *
     * @param aList the list in which to find the minimum
element.
     * @return the index of the minimum element in the list.
     */
    public int getMinIndex(ArrayList<String> aList) {
```

```java
        return getMinIndex(aList, 0);
    }


    /**
     * Finds the minimum element of a list from the startingIndex to the end
     * and returns the index of that element.
     * If there is more than one instance of the minimum,
     * then the lowest index will be returned.  Non-destructive.
     *
     * @param aList the list in which to find the minimum element.
     * @param startingIndex the integer for the starting index.
     * @return the index of the minimum element in the list.
     */
    private int getMinIndex(ArrayList<String> aList, int startingIndex) {
        if (isEnd(aList, startingIndex)) {
            return startingIndex;
        }

        else {
            if (aList.get(startingIndex).compareTo(getMin(aList, startingIndex + 1)) > 0) {
                return getMinIndex(aList, startingIndex + 1);
            }

            else {
                return startingIndex;
            }
        }
    }


    /**
     * Sorts a list in place. I.E. the list is modified so that it is in order.
     *
     * @param aList: the list to sort.
     */
    public void sort(ArrayList<String> aList) {
        sort(aList, 0);
    }


    /**
     * Sorts a list in place. I.E. the list is modified so
 that it is in order.
     *
     * @param aList the list in which to find the minimum
 element.
     * @param startingIndex the integer for the starting
 index.
     */
    private void sort(ArrayList<String> aList, int
 startingIndex) {
        if (! isEnd(aList, startingIndex)) {
            int minIndex = getMinIndex(aList, startingIndex);

            swap(aList, minIndex, startingIndex);
            sort(aList, startingIndex + 1);
        }
    }


    /**
     * Checks whether the index is at the end.
     *
     * @param aList the list to check
     * @prarm index the integer for the index
     */
    private boolean isEnd(ArrayList<String> aList, int index
 ) {
        if (index == (aList.size() - 1)) {
            return true;
        }

        else {
            return false;
            }
        }
}
```

```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",
                actual);

        Testing.assertEquals("getMin should not modify the list",
                originalList,
                copy);

        actual = lp.getMin(array2arraylist(new String[]{"aardvark", "lion", "zebra", "cougar", "cheetah"}));
        Testing.assertEquals("boundary case: minimum in first position",
                "aardvark",
                actual);

        actual = lp.getMin(array2arraylist(new String[]{"lion", "aardvark", "zebra", "cougar", "cheetah"}));
        Testing.assertEquals("boundary case: minimum in second position",
                "aardvark",
                actual);

        actual = lp.getMin(array2arraylist(new String[]{"lion", "zebra", "aardvark", "cougar", "cheetah"}));
        Testing.assertEquals("boundary case: minimum in third position",
                "aardvark",
                actual);

        actual = lp.getMin(array2arraylist(new String[]{"lion", "zebra", "cougar", "aardvark", "cheetah"}));
        Testing.assertEquals("boundary case: minimum in fourth position",
                "aardvark",
                actual);

        actual = lp.getMin(array2arraylist(new String[]{"bear", "lion", "zebra", "cougar", "antelope"}));
        Testing.assertEquals("boundary case: minimum in last position",
                "antelope",
                actual);
    }
```

```java

    public static void getMinIndexTests() {

        Testing.testSection("Testing getMinIndex");

        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);

        Testing.assertEquals("getMinIndex should return the index of the first occurrence of the min element",
                             2,
                             lp.getMinIndex(copy));

        Testing.assertEquals("getMinIndex should not modify the list",
                originalList,
                copy);

        int actual = lp.getMinIndex(array2arraylist
                (new String[]{"aardvark", "lion", "zebra", "cougar", "cheetah"}));
        Testing.assertEquals("boundary case: minimum in first position",
                0,
                actual);

        actual = lp.getMinIndex(array2arraylist(new String[]{"lion", "aardvark", "zebra", "cougar", "cheetah"}));
        Testing.assertEquals("boundary case: minimum in second position",
                1,
                actual);

        actual = lp.getMinIndex(array2arraylist(new String[]{"lion", "zebra", "aardvark", "cougar", "cheetah"}));
        Testing.assertEquals("boundary case: minimum in third
 position",
                2,
                actual);

        actual = lp.getMinIndex(array2arraylist(new String[]{
"lion", "zebra", "cougar", "aardvark", "cheetah"}));
        Testing.assertEquals("boundary case: minimum in
fourth position",
                3,
                actual);

        actual = lp.getMinIndex(array2arraylist
                (new String[]{"bear", "lion", "zebra", "
cougar", "antelope"}));
        Testing.assertEquals("boundary case: minimum in last
position",
                4,
                actual);

        actual = lp.getMinIndex(array2arraylist
                (new String[]{"antelope", "lion", "zebra", "
cougar", "antelope"}));
        Testing.assertEquals("boundary case: minimum in first
 and last position",
                0,
                actual);
    }

    public static void sortTests()
    {
        Testing.testSection("Testing sort");

        ListProcessor lp = new ListProcessor();

        String[] strings = {"b", "e", "a", "d", "g", "k", "c"
, "r", "t", "v", "a", "c", "b"};

        ArrayList<String> myList = array2arraylist(strings);

        lp.sort(myList);

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