

```
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 public class Testing {
12
13     private static boolean VERBOSE = false;
14     private static int numTests;
15     private static int numFails;
16
17     /**
18      * Toggles between a lot of output and little
19      * output.
20      *
21      * @param verbose
22      *           If verbose is true, then complete
23      *           information is printed,
24      *           whether the tests passes or fails. If
25      *           verbose is false, only
26      *           failures are printed.
27      */
28
29
30     /**
31      * Each of the assertEquals methods tests whether
```

```
31 the actual
32      * result equals the expected result. If it does,
33      then the test
34      *
35      * The only difference between these methods is the
36      types of the
37      *
38      * All take a String message and two values of some
39      other type to
40      *
41      * compare:
42      *
43      * @param message
44      *           a message or description of the test
45      * @param expected
46      *           the correct, or expected, value
47      * @param actual
48      *           the actual value
49
50
51      public static void assertEquals(String message,
52                                      boolean expected,
53                                      boolean actual)
54
55      {
56          printTestCaseInfo(message, "" + expected, "" +
57          actual);
58          if (expected == actual) {
59              pass();
60          } else {
61              fail(message);
62          }
63      }
64
65
66      public static void assertEquals(String message, int
67          expected, int actual)
68      {
69          printTestCaseInfo(message, "" + expected, "" +
70          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      * Asserts that a given boolean must be true.  The
97      * 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
101    * true.
102    */
103   public static void assertTrue(String message,
104     boolean actual)
105   {
106     assertEquals(message, true, actual);
107   }
108   /**
109    * Asserts that a given boolean must be false. The
110    * test fails if
111    * the boolean is not false (i.e. if it is true).
112    *
113    * @param message The test message
114    * @param actual The boolean value asserted to be
115    * false.
116    */
117   public static void assertFalse(String message,
118     boolean actual)
119   {
120     assertEquals(message, false, actual);
121   }
122   private static void printTestCaseInfo(String
123     message, String expected,
124                                         String
125     actual)
126   {
127     if (VERBOSE) {
128       System.out.println(message + ":");
129       System.out.println("expected: " + expected
130 );
131       System.out.println("actual: " + actual);
132     }
133   }
```

```
127      }
128
129      private static void pass()
130      {
131          numTests++;
132
133          if (VERBOSE) {
134              System.out.println("");
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("");
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
171   * before running any
172   * tests, so that passes and fails are correctly
173   * tallied.
174   */
175  public static void startTests()
176  {
177      System.out.println("Starting Tests");
178      System.out.println();
179      numTests = 0;
180      numFails = 0;
181  }
182  /**
183   * Prints out summary data at end of tests.
184   * Should be called
185   * after all the tests have run.
186   */
187  public static void finishTests()
188  {
189      System.out.println("=====");
190      System.out.println("Tests Complete");
191      System.out.println("=====");
192      int numPasses = numTests - numFails;
193
194      System.out.print(numPasses + "/" + numTests +
195          " PASS ");
196      System.out.printf("(pass rate: %.1f%s)\n",
197          100 * ((double) numPasses
198          ) / numTests,
199          "%");
```

```
196
197     System.out.print(numFails + "/" + numTests +
198             " FAIL ");
199     System.out.printf("(fail rate: %.1f%s)\n",
200                     100 * ((double) numFails) /
201                     numTests,
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
40}
```

```
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
67   * list.
68   *
69   * @param aList the list to search
70   * @return the index of the first occurrence of the
71   * minimum element
72   */
73  public int getMinIndex(ArrayList<String> aList)
74  {
75    return getMinIndex(aList, 0);
76  }
77
78  /**
79   * Sorts a list in place. I.E. the list is modified
80   * so that it is in order.
81   *
82   * @param aList: the list to sort.
83   */
84  private void sort(ArrayList<String> aList, int
85 startingIndex)
86  {
87    if (startingIndex >= aList.size() - 1) {
88      return;
89    }
90
91    swap(aList, getMinIndex(aList, startingIndex),
92         startingIndex);
93
94    sort(aList, startingIndex + 1);
95  }
96
97  /**
98   * Sorts the list in alphabetical order using
99
```

```
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        public static void getMinTests() {
25            Testing.testSection("Testing getMin");
26
27            ListProcessor lp = new ListProcessor();
28
29            String[] strings = {"b", "e", "a", "d", "g", "k",
30                "c", "r", "t", "v", "a", "c", "b"};
31            ArrayList<String> originalList =
32                array2arraylist(strings);
33            ArrayList<String> copy = new ArrayList<String>(
34                originalList);
35            // makes a copy of originalList
36
37            String actual = lp.getMin(copy);
```

File - C:\Users\james\Documents\Personal\College Life\Second Year\Courses\CSC\CSC120\Lab9\Lab09 starter code\src\

```
34         Testing.assertEquals("The minimum of a list of
35             strings is the first in alphabetical order",
36                 "a",
37                 actual);
38
39         Testing.assertEquals("getMin should not modify
40             the list",
41                 originalList,
42                 copy);
43
44         actual = lp.getMin(array2arraylist(new String
45             []{"aardvark", "lion", "zebra", "cougar", "cheetah"}));
46         Testing.assertEquals("boundary case: minimum in
47             first position",
48                 "aardvark",
49                 actual);
50
51         actual = lp.getMin(array2arraylist(new String
52             []{"bear", "lion", "zebra", "cougar", "antelope"}));
53         Testing.assertEquals("boundary case: minimum in
54             last position",
55                 "antelope",
56                 actual);
57     }
58
59
60     public static void getMinIndexTests() {
61
62         Testing.testSection("Testing getMinIndex");
63
64         ListProcessor lp = new ListProcessor();
65         String[] strings = {"b", "e", "a", "d", "g", "k",
66             "c", "r", "t", "v", "a", "c", "b"};
67         ArrayList<String> originalList =
68             array2arraylist(strings);
69         ArrayList<String> copy = new ArrayList<String>(
70             originalList);
71
72         Testing.assertEquals("getMinIndex should return the
73             index of the minimum element in the list",
74                 2,
75                 lp.getMinIndex(copy));
76
77         Testing.assertEquals("getMinIndex should not modify
78             the list",
79                 originalList,
80                 copy);
81
82         Testing.assertEquals("getMinIndex should return the
83             index of the minimum element in the list",
84                 2,
85                 lp.getMinIndex(originalList));
86
87         Testing.assertEquals("getMinIndex should not modify
88             the list",
89                 originalList,
90                 copy);
91
92         Testing.assertEquals("getMinIndex should return the
93             index of the minimum element in the list",
94                 2,
95                 lp.getMinIndex(array2arraylist(strings)));
96
97         Testing.assertEquals("getMinIndex should not modify
98             the list",
99                 originalList,
100                copy);
101    }
```

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