```java
import static org.junit.Assert.assertEquals;

/**
 * JUnit test fixture for {@code Map<String, String>}'s constructor and kernel
 * methods.
 *
 * @author Put your name here
 *
 */
public abstract class MapTest {

    /**
     * Invokes the appropriate {@code Map} constructor for the implementation
     * under test and returns the result.
     *
     * @return the new map
     * @ensures constructorTest = {}
     */
    protected abstract Map<String, String> constructorTest();

    /**
     * Invokes the appropriate {@code Map} constructor for the reference
     * implementation and returns the result.
     *
     * @return the new map
     * @ensures constructorRef = {}
     */
    protected abstract Map<String, String> constructorRef();

    /**
     *
     * Creates and returns a {@code Map<String, String>} of the implementation
     * under test type with the given entries.
     *
     * @param args
     *            the (key, value) pairs for the map
     * @return the constructed map
     * @requires <pre>
     * [args.length is even]  and
     * [the 'key' entries in args are unique]
     * </pre>
     * @ensures createFromArgsTest = [pairs in args]
     */
    private Map<String, String> createFromArgsTest(String... args) {
        assert args.length % 2 == 0 : "Violation of: args.length is even";
        Map<String, String> map = this.constructorTest();
        for (int i = 0; i < args.length; i += 2) {
            assert !map.hasKey(args[i]) : ""
                    + "Violation of: the 'key' entries in args are unique";
            map.add(args[i], args[i + 1]);
        }
        return map;
    }

    /**
     *
     * Creates and returns a {@code Map<String, String>} of the reference
```

```
 62      * implementation type with the given entries.
 63      *
 64      * @param args
 65      *            the (key, value) pairs for the map
 66      * @return the constructed map
 67      * @requires <pre>
 68      * [args.length is even]  and
 69      * [the 'key' entries in args are unique]
 70      * </pre>
 71      * @ensures createFromArgsRef = [pairs in args]
 72      */
 73     private Map<String, String> createFromArgsRef(String... args) {
 74         assert args.length % 2 == 0 : "Violation of: args.length is even";
 75         Map<String, String> map = this.constructorRef();
 76         for (int i = 0; i < args.length; i += 2) {
 77             assert !map.hasKey(args[i]) : ""
 78                     + "Violation of: the 'key' entries in args are unique";
 79             map.add(args[i], args[i + 1]);
 80         }
 81         return map;
 82     }
 83
 84     // CONSTRUCTORS ------------------------------------------------------------
 85
 86     /**
 87      * Test for no argument constructor case.
 88      */
 89     @Test
 90     public final void noArgConstructorTest() {
 91         // Setup
 92         Map<String, String> m = this.createFromArgsTest();
 93         Map<String, String> mExpected = this.createFromArgsRef();
 94
 95         // Eval
 96         assertEquals(mExpected, m);
 97     }
 98
 99     /**
100      * Test for single pair argument constructor case.
101      */
102     @Test
103     public final void singleArgConstructorTest() {
104         // Setup
105         Map<String, String> m = this.createFromArgsTest("1", "a");
106         Map<String, String> mExpected = this.createFromArgsRef("1", "a");
107
108         // Eval
109         assertEquals(mExpected, m);
110     }
111
112     /**
113      * Test for 2 pairs argument constructor case.
114      */
115     @Test
116     public final void twoArgConstructorTest() {
117         // Setup
118         Map<String, String> m = this.createFromArgsTest("1", "a", "2", "b");
```

```
119         Map<String, String> mExpected = this.createFromArgsRef("1", "a", "2",
120                 "b");
121
122         // Eval
123         assertEquals(mExpected, m);
124     }
125
126     // KERNEL ------------------------------------------------------------------
127     // add ---------------------------------------------------------------------
128     /**
129      * Test adding a single k v pair to an empty map.
130      */
131     @Test
132     public final void testAddSinglePairToEmpty() {
133         // Setup
134         Map<String, String> m = this.createFromArgsTest();
135         Map<String, String> mExpected = this.createFromArgsRef("1", "a");
136         // Call
137         m.add("1", "a");
138         // Eval
139         assertEquals(mExpected, m);
140     }
141
142     // remove ------------------------------------------------------------------
143     /**
144      * Test removing a single k v pair resulting in empty map.
145      */
146     @Test
147     public final void testRemoveSinglePairToEmpty() {
148         // Setup
149         Map<String, String> m = this.createFromArgsTest("1", "a");
150         Map<String, String> mExpected = this.createFromArgsRef();
151         Map.Pair<String, String> pair;
152         // Unable to declare a map.pair object without simplepair
153         //  messy workaround
154         Map<String, String> pairExpected = this.createFromArgsRef("1", "a");
155         // Call
156         pair = m.remove("1");
157         // Eval
158         assertEquals(pairExpected.remove("1"), pair);
159         assertEquals(mExpected, m);
160     }
161
162     // removeAny ---------------------------------------------------------------
163     /**
164      * Test removing any k v pair
165      */
166     @Test
167     public final void testRemoveAny() {
168         // Setup
169         Map<String, String> m = this.createFromArgsTest("1", "a", "1", "b");
170         Map<String, String> mExpected = this.createFromArgsRef("1", "a");
171         Map<String, String> mExpected2 = this.createFromArgsRef("1", "b");
172         //create boolean to check if the pair removed was in m
173         boolean goodRemove = false;
174         // Call
175         m.removeAny();
```

```
176         if (m.equals(mExpected) || m.equals(mExpected2)) {
177             goodRemove = true;
178         }
179         // Eval
180         assertEquals(true, goodRemove);
181     }
182     // value -----------------------------------------------------------------
183     /**
184      * Test reporting the value v at key k.
185      */
186     @Test
187     public final void testValue() {
188         // Setup
189         Map<String, String> m = this.createFromArgsRef("1", "a");
190         // Call
191         String value = m.value("1");
192         // Eval
193         assertEquals("a", value);
194     }
195     // hasKey ----------------------------------------------------------------
196     /**
197      * Test removing a single k v pair resulting in empty map.
198      */
199     @Test
200     public final void testHasKey() {
201         // Setup
202         Map<String, String> m = this.createFromArgsRef("1", "a");
203         // Call
204         boolean trueKey = m.hasKey("1");
205         boolean falseKey = m.hasKey("2");
206         // Eval
207         assertEquals(true, trueKey);
208         assertEquals(false, falseKey);
209     }
210     // size ------------------------------------------------------------------
211     /**
212      * Test reporting the size of a map.
213      */
214     @Test
215     public final void testMapSize() {
216         // Setup
217         Map<String, String> m0 = this.createFromArgsRef();
218         Map<String, String> m1 = this.createFromArgsRef("1", "a");
219         Map<String, String> m2 = this.createFromArgsRef("1", "a", "2", "b");
220         int zeroRef = 0;
221         int oneRef = 1;
222         int twoRef = 2;
223
224         // Call
225         int zero = m0.size();
226         int one = m1.size();
227         int two = m2.size();
228
229         // Eval
230         assertEquals(zeroRef, zero);
231         assertEquals(oneRef, one);
232         assertEquals(twoRef, two);
```

```
233     }
234
235
236     /**
237      * Test size = 0;
238      */
239     @Test
240     public final void testSizeEmpty() {
241         // Setup
242         Map<String, String> m = this.createFromArgsTest();
243         Map<String, String> mExpected = this.createFromArgsRef();
244         // Call
245         // Eval
246         assertEquals(mExpected.size(), m.size());
247     }
248
249     /**
250      * Test size = 1;
251      */
252     @Test
253     public final void testSize1() {
254         // Setup
255         Map<String, String> m = this.createFromArgsTest("1", "a");
256         Map<String, String> mExpected = this.createFromArgsRef("1", "a");
257         // Call
258         // Eval
259         assertEquals(mExpected.size(), m.size());
260     }
261
262     /**
263      * Test size = 0 after -1 using remove();
264      */
265     @Test
266     public final void testSize1minus1() {
267         // Setup
268         Map<String, String> m = this.createFromArgsTest("1", "a");
269         Map<String, String> mExpected = this.createFromArgsRef();
270         // Call
271         m.remove("1");
272         // Eval
273         assertEquals(mExpected.size(), m.size());
274     }
275
276     /**
277      * Test size = 0 after -1 using removeAny();
278      */
279     @Test
280     public final void testSize1minusAny() {
281         // Setup
282         Map<String, String> m = this.createFromArgsTest("1", "a");
283         Map<String, String> mExpected = this.createFromArgsRef();
284         // Call
285         m.removeAny();
286         // Eval
287         assertEquals(mExpected.size(), m.size());
288     }
289
```

```
290 }
291
```