solution

Imagine that we are to implement a List with a linked list and that we want to write some tests for a `set` method whose definition is below:

```
/**
 * Overwrites the element located at the given index with the new value.
 *
 * @throws IndexOutOfBoundsException if the index < 0 or index >= this.size()
 */
public void set(int index, int value);
```

In this exercise, we'll be writing tests before writing the implementation for this method. These types of tests are called black-box tests, as the implementation details are like a black-box we can't see inside. So instead of relying on any knowledge of how the code is implemented, we'll write tests based on the specification requirements above (in your homework, this will be in comments in the code and on the website homework assignment pages).

Take some time to try and come up with test cases or things to check based on the specification. Reminder: test cases are about checking that the result is what you expect, so it is often helpful to frame your test case ideas as: "in X situation, make sure that Y happens/doesn't happen/is true/is false".

Here are some possible test cases or things to check:

solution:

- Make sure that an exception is thrown if index < 0 **(Forbidden Input)**
- Make sure that an exception is thrown if index $\geq$ this.size() **(Forbidden Input)**
- Make sure that the old element at the given index is gone and replaced with the new item **(Expected Behavior)**
- Make sure that the other elements not at the given index are untouched (you generally can just combine this and the above one to happen in every test, see below **(Expected Behavior)**
- Make sure that different valid positions for the index (front, middle, back) still `set`s properly **(Boundary/Edge Cases)**
- Make sure that different valid values for the 'value' parameter (seems like anything is valid) still `set`s properly **(Expected Behavior)**
- Make sure that that exceptions are not thrown for valid indices (we'll see that normally if exceptions are thrown, then tests will fail by default - so we won't need to explicitly add a test for this, because the other general tests will cover it.) **(Expected Behavior)**
- Make sure that general cases work for different sizes **(Scale)**

Now let's try implementing some of these as JUnit tests.
As mentioned earlier, the point of tests is to assert that the code meets our expectations for the behavior. So the general structure of a JUnit test for method A will be something like:

1. set up your data structure / object
2. call method A
3. assert the result/effect of method A is what you expect

Typically, we will use a method called `assertThat(actual, matcher)` and pass in 2 parameters: the actual is going to be the thing we want to check (a data structure, a return value from a method call, a variable, etc.) and the matcher represents what we want to check about it (usually specified by calling a method). Below are a couple of general examples to clarify:

```
    assertThat(5 * 2, is(10));
    assertThat("CSE 373 rocks?", containsString("373");

    /** for TAs, the equivalent assertEquals/ other ways of doing it **/
    assertEquals(10, 5 * 2);
```

Note: these tests are just trivial examples of assertThat – next we'll go over some examples of actually using assertThat to test our method behavior instead of math or String properties.

```
        @Test
        public void testSetGeneral() {
            LinkedIntList list = new LinkedIntList(new int[]{2, 4, 6, 8, 10});
            list.set(2, 100);

            assertThat(list.get(2), is(100));

            assertThat(list.get(0), is(2));
            assertThat(list.get(1), is(4));
            assertThat(list.get(3), is(8));
            assertThat(list.get(4), is(10));

            assertThat(list.size(), is(5));
        }

        @Test
        public void testSetFront() {
            LinkedIntList list = new LinkedIntList(new int[]{2, 4, 6, 8, 10});
            list.set(0, 100);

            assertThat(list.get(0), is(100));

            assertThat(list.get(1), is(4));
            assertThat(list.get(2), is(6));
            assertThat(list.get(3), is(8));
            assertThat(list.get(4), is(10));

            assertThat(list.size(), is(5));
        }

        @Test
        public void testSetBack() {
            LinkedIntList list = new LinkedIntList(new int[]{2, 4, 6, 8, 10});
            list.set(4, 100);

            assertThat(list.get(4), is(100));

            assertThat(list.get(0), is(2));
            assertThat(list.get(1), is(4));
            assertThat(list.get(2), is(6));
            assertThat(list.get(3), is(8));

            assertThat(list.size(), is(5));
        }

        @Test
        public void testSetNegativeIndexThrowsException() {
            LinkedIntList list = new LinkedIntList(new int[]{2, 4, 6, 8, 10});
            assertThrows(IndexOutOfBoundsException.class, () -> { list.set(-1, 100); });
        }

        @Test
        public void testSetTooLargeIndexThrowsException() {
            LinkedIntList list = new LinkedIntList(new int[]{2, 4, 6, 8, 10});
            assertThrows(IndexOutOfBoundsException.class, () -> { list.set(100, 100); });
        }
```

You might notice `is()` is used a lot - whenever you want to check a simple value, you'll probably want to use `is()` as part of the last parameter.

Typically the way you write tests is copy structure from existing / given tests and then change them as you realize what differences you want.

How do all the multiple calls to assertThat work in a single test? Every time assertThat is called, we're making sure that something else meets our expectation. If our expectations are not met and the assertion isn't true (e.g. list.get(0) is actually not 100), then an AssertionError will be thrown and the test fails. In IntelliJ, this will be represented with a red circled X next to the specific test name that failed. If an assertThat method call passes, however, it'll just move on to the next line of code. If the test method finishes, then the test passes and will be represented with a green circled check mark next to the test that passed.
Here's a buggy implementation of set for a LinkedIntList that works in some cases.

```
public void set(int index, int value) {
    ListNode current = front;
    for (int i = 0; i < index - 1; i++) {
        current = current.next;
    }
    ListNode newNode = new ListNode(value, current.next.next);
    current.next = newNode;
}
```

Imagine we knew this was the implementation - let's run our tests against this!
Another thing you can do is called clear box testing: try to look at the code now and find situations it could break! Look for if/elses that you should test out, any places that have complicated logic and write tests that attempt to trigger the complicated code. In general try triggering code that has to do with different combinations of entering if/elses, entering/not entering loops, etc.

Here the bugs is that `set` doesn't throw exceptions (could be caught by black box testing), and they don't handle if the front needs to be updated (could be caught by clear-box or black-box testing).