```
// This is a method that tests onlyEnglishLetters
@Test
public void testOnlyEnglishLetters () {
  assertEquals (false, HW2.onlyEnglishLetters("")); // Test none
  assertEquals (true, HW2.onlyEnglishLetters("a")); // Test one
  assertEquals (false, HW2.onlyEnglishLetters("1!")); // Test two?
  assertEquals (true, HW2.onlyEnglishLetters("abDkfdoFRs")); // Test many
  assertEquals (false, HW2.onlyEnglishLetters("lFKe5aaa"));
}
```

In this method, I interpreted test none, one, and many as that many iterations. So my first test was with an empty, non-existent, string, the second test was with an english character and a non-english character, and the last one was a mix of both. These tests were successful, as each of the expected values worked.

```
@Test
// This is a method that tests if the method being tested will replace the correct index
public void testReplaceKth () {
  assertEquals ("", HW2.replaceKth('a', 'x', 3, "")); // Test none
  assertEquals ("abcaxa", HW2.replaceKth('a', 'x', 3, "abcaaa")); // Test middle
  assertEquals ("aaaaa",HW2.replaceKth('a', 'x', 6, "aaaaa")); // Test last (/ none)
}
```

In this method, I used test first, last, and middle, but interpreted test first as test none. If the string was empty, it should have no inputs of the replacement character and if there were too few inputs for the string, it should have no replacements. These tests were successful as each of the expected values worked.

```
@Test
// This is a method that tests if the method being tested with intertwine the two strings correctly
public void testInterleave () {
  assertEquals ("AaBbCcde", HW2.interleave("ABC", "abcde")); // Test first
  assertEquals ("aAbBcCde", HW2.interleave("abcde", "ABC")); // Test second
}
```

In this method, I tested if the code would work if the first or the second string was the longest, which is how I tested the “first” and “last” iterations. They both intertwined correctly, and each of the tests were successful.

```
@Test
// This is a method that tests if the blank words method will successfully create blank words
public void testBlankWords () {
  assertEquals ("", HW2.blankWords("")); // Test none
  assertEquals ("T", HW2.blankWords("T")); // Test one
  assertEquals ("T__s is a T__t.", HW2.blankWords("This is a Test.")); //Test many
}
```

In this method, I tested if the method would work with a “none, one, many” test. I interpreted this test as the amount of times the string ran, so the first one was no string (so no iterations), the second one

was one letter, so one iteration, and the last one was a long string with multiple iterations. All of the tests worked as expected.

```
@Test
// This is a method that tests if the nth words of a string sequence will become a new sequence
public void testNthWord () {
  assertEquals("", HW2.nthWord(1, "")); // Test none
  assertEquals("zero ", HW2.nthWord(10, "zero one    two  three four five six seven")); // Test one
  assertEquals("zero three six " ,HW2.nthWord(3, "zero one    two  three four five six seven")); // Test many
}
```

In this method, I tested a “none, one, many” test where I tested none with an empty string, I tested one with only relaying out the first word in a long string (since the amount of words that were in the string were longer than the iterations for the nth word), and then I tested if multiple words would work. Technically the tests all worked, but also I couldn’t figure out how to not add a space at the end of the string.

```
@Test
// This is a method that tests if a string will truncate at a given "truncation" spot at/after the index
public void testTruncateAfter () {
  assertEquals("La-", HW2.truncateAfter(1,"La-te-ly the-re.")); // Test first
  assertEquals("Late-", HW2.truncateAfter(5, "La-te-ly the-re.")); // Test middle
  assertEquals("Lately the-", HW2.truncateAfter(7, "La-te-ly the-re.")); // Test middle -- fail
  assertEquals("Lately", HW2.truncateAfter(6, "La-te-ly the-re.")); // Test middle
  assertEquals("Lately there.", HW2.truncateAfter(20,"La-te-ly the-re.")); // Test last
}
```

In this method, I tested the truncation after by testing if the first truncation spot would work, if the middle truncation spots would work (both at the spot and after) and if there was no truncation spot it would just run all the way through. It technically worked at all of the spots except for index 7, since the string wanted to truncate at the space rather than the hyphen.