Project 2 Testing

This testing document was written by making use of the markdown editor *Obsidian*. I apologize in advance for potentially strange formatting, but I tried my best to organize the sections of this document so that they are aligned with the assertions in my testing file "HW2Tests.java".

My Methodology

I worked through all of these tests making use of the Test 0, Test 1, Test many methodology outlined in the project instructions. In some cases, I tested negative values in order to ensure the effectiveness of my code. I will indicate which tests I am referring to thorugh descriptions at the start of my bulletted lists. As per the instrcutions of the project, I will not be including the actual code in this document.

As with the last project, I used the following imports:

```
import org.junit.*;
import static org.junit.Assert.*;
```

Organization:

I included code snippets directly from my java file to aid the reader. Each code snippet is explained in the bullet point directly below the code.

Instruction's Examples

I tested all of the examples from the instructions and provided brief explanations as well.

Fields

There are no fields in this project's testing file!

Testing the Actual Methods

---

testSamePrefix

```
// Test string lengths of 0
assertTrue(HW2.samePrefix("","",0));

// Test string lengths of 0 with comparison 1
assertFalse(HW2.samePrefix("","",1));

// Test string lengths of 0 with large comparison
assertFalse(HW2.samePrefix("","",100));
```

- I first started comparing two empty strings, first with a 0 comparison, then 1, then 100. The comparison of 0 on empty strings should return true as there is the same nothing being compared against themselves. Testing 1 and 100 both should return false because of the criteria that comparing strings with an index greater than the string length is always false. These behaviors are confirmed by these tests.

```
// Test a character amount of 1 with identical first letters
assertTrue(HW2.samePrefix("h", "h", 0));

// Test a character amount of 1 with different first letters and 0 comparison
assertTrue(HW2.samePrefix("m", "h", 0));

// Test a character amount of 1 with different first letters and negative
comparison
assertTrue(HW2.samePrefix("m", "h", -1));
```

- Then, I tested two more cases of comparison 0 and one with a negative comparison. Since a comparison of 0 or a negative value isn't actually comparing anything in the actual strings, true is returned by default by design. This is observed regardless of the strings inputted The three tests here all return true, proving that this behavior works as intended.

```
// Test a character amount of 1 with identical first letters
assertTrue(HW2.samePrefix("h", "h", 1));

// Test a character amount of 1 with different first letters
assertFalse(HW2.samePrefix("H","h",1));
```

- Then, I tested strings of length 1, comparing only that one character. The first test confirmed a true result of "h vs. h", while the second confirmed a false result of "h vs. H". By design, the letters case is necessary when making a comparison, which is why these tests work and are useful to this document.

```
// Test a character amount of 4 with identical strings
assertTrue(HW2.samePrefix("this", "this", 4));

// Test a character amount of 4 with different strings
assertFalse(HW2.samePrefix("This", "this", 4));
```

- I then compared multiple character length strings. First I compared "this" with "this" at an index of 4. Since "this" is 4 characters long, the result should be true, as confirmed by the test. Then I compared "this" with "This". This gave a false output because of the difference in case of the first letter.

```
// Test a character amount greater than the length of one string
assertFalse(HW2.samePrefix("this", "this is", 5));

// Test a character amount much greater than both strings
assertFalse(HW2.samePrefix("this", "this", 100));
```

- Then, I compared an inputted comparison amount that exceeded only one of the strings lengths. In this case, false is returned by default because of the length criteria listed in my javadoc, as confirmed by the test. I then tested the amount being greater than both strings, which also returned false as expected.

```
// Test strings that are identical up to the integer inputted
assertTrue(HW2.samePrefix("this test", "this trial", 6));
```

```
// Test strings that differ at the integer inputted
assertFalse(HW2.samePrefix("this test", "this trial", 7));
```

- I then tested longer phrases, with the first comparison cutting off at the final equivalent character, and the next cutting off directly after this cutoff. These results should be true and false, respectively. I observed this behavior using the assertions for boolean values.

```
// Test the first example from the instructions
assertTrue(HW2.samePrefix("this is a test", "this is a trial", 11));

// Test the second example from the instructions
assertFalse(HW2.samePrefix("this is a test", "this is a trial", 12));

// Test the third example from the instructions
assertFalse(HW2.samePrefix("this is a test", "This is a trial", 4));

// Test the fourth example from the instructions
assertFalse(HW2.samePrefix("this is a test", "this is a test", 100));
```

- I then tested the 4 examples from the instructions. These instructions laid out the basics of the tests above, and running them through the assert (expecting the result given in the document) proved that my code works for these cases.

---

testMatchingParentheses

```
// Test an empty string
assertTrue(HW2.matchingParentheses(""));

// Test a string of length 1
assertTrue(HW2.matchingParentheses("i"));

// Test a string of length one with only (
assertFalse(HW2.matchingParentheses("("));
```

```
// Test a string of length one with only )
assertFalse(HW2.matchingParentheses(")"));
```

- I grouped the test 0 and test 1 parts of the requirements together in this case. If there are no parentheses found, or the string is empty, then True is returned by default as asserted by the tests. The next two assertions involve only one character, being either an opening or closing parentheses. These should both be false because there cannot be matched parentheses when only one of them is found.

```
// Test a string that has no parentheses
assertTrue(HW2.matchingParentheses("this is a test!"));

// Test a string with one set of matching parentheses
assertTrue(HW2.matchingParentheses("(this) is a test"));

// Test a string with one set of mismatching parentheses
assertFalse(HW2.matchingParentheses("this )is( a test"));

// Test a string with multiple sets of matching parentheses
assertTrue(HW2.matchingParentheses("(this) is a ((test) of parentheses)"));
```

- I then tested 0, 1, many in respect to the number of matched parentheses in my input. The first one should be true as there are no parentheses in the string. The second test should be true as well because there is one pair of matched parentheses. The third test should return false because even though there is a pair of parentheses, they are in the incorrect order. The fourth test here tests many parentheses. There are 3 pairs of parentheses here, all correctly order despite one being nested. This resulted in a correctly asserted true return.

```
// Test a string with closing parentheses first
assertFalse(HW2.matchingParentheses(") (wow!)"));

// Test a string with all matching parentheses besides a ( in the middle
assertFalse(HW2.matchingParentheses("(this) is a ((test) of parenthe(ses)"));

// Test a string with all matching parentheses besides a ) in the middle
assertFalse(HW2.matchingParentheses("(this) is a ((test) of parenthe)ses)"));
```

```
// Test a string with opening parentheses last
assertFalse(HW2.matchingParentheses("(wow!) ("));
```

- Then, I tested first, middle, last in respect to where I placed the mismatched parenthesis. In the first test, I started the string with a closing bracket, which should be false by default. This same result is expected from an open parenthesis at the final index of the string. This is because there cannot be (at the end or) at the beginning while meeting the criteria of being matched. As for the two middle tests, I placed the mismatched parenthesis in the middle of the string. Since they were present here, the result should be false, as confirmed by my assert statements.

```
// Test a string with only parentheses, that are matched
assertTrue(HW2.matchingParentheses("(((())))"));

// Test a string with only parentheses, that are unmatched
assertFalse(HW2.matchingParentheses("((()))"));
```

- I then tested strings that consisted only of parentheses to make sure I covered all my bases. There are an even and matched number of parentheses in the first assertion, which is why the result is true. Since there are aren't a closing parentheses proceeded by a corresponding open parenthesis in the second string, false is returned.

```
// Test the first example from the instructions
assertTrue(HW2.matchingParentheses("This is a (test (of the) (matching))
parentheses"));

// Test the first example from the instructions
assertFalse(HW2.matchingParentheses("The (second closing) parenthesis) does not
match"));
```

- Finally, I tested the examples from the instructions. The instructions listed two examples. One where there is a sentence with matched parentheses and one where there isn't. Examining these strings with my method reveals the behavior of the parentheses in these two strings, where true is asserted for the first and false for the second.

---

testRemoveEveryKthWord

```
// Test an empty string with a negative k
assertEquals("", HW2.removeEveryKthWord("", -10));

// Test an empty string
assertEquals("",HW2.removeEveryKthWord("", 0));

// Test an empty string with k = 1
assertEquals("", HW2.removeEveryKthWord("", 1));

// Test an empty string with a positive k
assertEquals("",HW2.removeEveryKthWord("", 100));
```

- First I tested an empty string with different k values. In all of these cases, an empty string should be returned because there were no inputted characters in the original argument. K values in the negatives, 0, 1, and many all outputted the same empty string that I started with, guaranteeing the efficacy of these tests.

```
// Test a single character string with a negative k
assertEquals("h", HW2.removeEveryKthWord("h", -10));

// Test a single character string with 0 k
assertEquals("h",HW2.removeEveryKthWord("h", 0));

// Test a single character string with k = 1, removing every single character
assertEquals("", HW2.removeEveryKthWord("h", 1));

// Test a single character string with a positive k
assertEquals("h",HW2.removeEveryKthWord("h", 100));
```

- I then tested a string with only 1 character, following the same test 0, 1, many rule for the k value. Negative, Zero, and too large of k values should have no impact on the inputted string, as observed by the assertions here. when k = 1, however, every single character in the argument should be removed, returning an empty string.

```
// Test a long string with a negative k
assertEquals("Testing this method", HW2.removeEveryKthWord("Testing this
method", -10));

// Test a long string with a zero k
assertEquals("Testing this method", HW2.removeEveryKthWord("Testing this
method", 0));

// Test a long string with k = 1, removing every single word
assertEquals("", HW2.removeEveryKthWord("Testing this method", 1));

// Test a long string with k > number of words, changing nothing
assertEquals("Testing this method", HW2.removeEveryKthWord("Testing this
method", 100));
```

- To satisfy the test many part of the string, I used the sentence "Testing this method" to perform the following tests. I performed the same test 0, 1, many for the k value here. I expected the only real change to occur with k = 1, where an empty string would be returned. This was the case, as confirmed by the corresponding assertions where if $\forall k \in \{-10, 0, 100\}$ did not make any changes to the input.

```
// Tests removing the last word in a string
assertEquals("This is a test ", HW2.removeEveryKthWord("This is a test yeah!",
5));

// Test removing every 4 in a string with 8 words
assertEquals("This is such a test wow! ", HW2.removeEveryKthWord("This is such
womp a test wow! womp", 4));
```

- The next test I performed was taking specific words out of a string. Removing the final word in a 5 word sentence should strip the string down of only that word, leaving behind the space before it. The first of theses two tests confirmed this behavior. For the second test here, I wanted to remove every 4th word of an 8 word string, leaving in spaces similar to that described above. This was observed as well by the positive assertion statement.

```
// Test if trailing spaces are dropped completely
assertEquals("This is a funky little trial",
```

```
        HW2.removeEveryKthWord("This womp is womp a womp funky womp little womp
trial        ", 2));
```

- Then, I decided to test two more edge cases. I accomplished this in one test by testing a string that had lots of trailing spaces, which should be dropped, as well as a number of words that was not a multiple of the k value. Due to my creation of the strTrim helper method, any trailing zeros are trimmed to account for any potential errors. I used the word "womp" as the one I expected to be removed, and these behaviors were confirmed by my assert statement.

```
// Test the first example from the instructions
assertEquals("Four score seven years our fore ",
        HW2.removeEveryKthWord("Four score and seven years ago our fore
fathers", 3));

// Test the second example from the instructions - leading zeros
assertEquals(" Every down Whoville Christmas lot!",
        HW2.removeEveryKthWord(" Every Who down in Whoville liked Christmas a
lot!", 2));
```

- Finally, I tested the two examples from the instructions. The first example was nothing special, and I expected the result based on the success of my previous tests of this method. I knew the second test would work because of the fact that I made my code include any leading spaces in the output. I only did this for leading spaces because of this example, and it was never indicated that this behavior must be followed by trailing spaces. The result s given in the instructions aligned with those returned by my method, indicating correct code.

---

testFlipEachK

```
// Test an empty string with negative k
assertEquals("",HW2.flipEachK("", -10));

// Test an empty string with k = 0
assertEquals("", HW2.flipEachK("", 0));

// Test an empty string with k = 1
```

```
assertEquals("", HW2.flipEachK("", 1));

// Test an empty string with a large k
assertEquals("", HW2.flipEachK("", 100));
```

- I tested an empty string with negative, 0, 1 and large values for my k here. Because there is nothing contained in this string, the output will also be an empty string, regardless of the k value. This is observed as seen above.

```
// Test a single character string with negative k
assertEquals("h",HW2.flipEachK("h", -10));

// Test a single character string with k = 0
assertEquals("h", HW2.flipEachK("h", 0));

// Test a single character string with k = 1
assertEquals("h", HW2.flipEachK("h", 1));

// Test a single character string with a k larger than the length of the string
assertEquals("h", HW2.flipEachK("h", 100));
```

- I tested a string with only one character to fulfil the test 1 branch of the string tests. I repeated the negative, 0, 1, and large k values with this single character. Since there is nothing to flip in the string when there is only one character, the inputted string is returned, which is observed in the tests seen above.

```
// Test a single word string with negative k
assertEquals("this",HW2.flipEachK("this", -10));

// Test a single word string with k = 0
assertEquals("this", HW2.flipEachK("this", 0));

// Test a single word string with k = 1
assertEquals("this", HW2.flipEachK("this", 1));
```

- I then tested a longer string to fulfil the "many" branch of the string testing. In these three tests seen above, I tested negative, 0, and 1 values for k. The output here should not be

changed in comparison to the input because of how the method works. This is indeed observed and the tests above confirm this.

```
// Test a single word string with a k equal to the string length
assertEquals("this", HW2.flipEachK("this", "this".length()));

// Test a single word string with a k half of the string length
assertEquals("thsi", HW2.flipEachK("this", "this".length() / 2));

// Test a single word string with a k larger than the length of the string
assertEquals("this", HW2.flipEachK("this", 100));
```

- I tested the same string as above (this) with 2 realistic k values as well as one that is too large for the given string. First, I used a k value equal to the length of the inputted string, which would result in an unchanged output as the method preserves the first k characters.
- Next, I tested a k value half of the inputted string, which should result in the first 2 characters being preserved then the next two flipped, or "thsi" in this case. This is confirmed here.
- Finally, I tested a k value of 100 as it exceeds the length of the strings. All three of these tests were successful, indicating correct code.

```
// Test the string from the instructions to test a k such that
statement.length() mod k != 0.
assertEquals("abcdhgfeijklnm", HW2.flipEachK("abcdefghijklmn", 4));

// Test a very long string with a k such that statement.length() mod k != 0.
assertEquals("Thii ss ats rest sestfo  th simetdoh inht e socend orpjecf tor
DSCS132",
        HW2.flipEachK("This is a stress test of this method in the second
project for CSDS132", 3));
```

When I say statement.length() mod k != 0, I mean there are a non-divisible number of characters compared to the inputted k value

- The first test here is taken straight from the instructions. This example is good to include because there are two many characters to be evenly divided by the k value, leaving some leftover. This should not truncate the string, but instead preserve these rolled-over characters. This correctly occurs in the first test.

- Next, I tested a very long string whose characters are not divisible by the k value. This resulted in a complex output that I placed in the expected part of the assertion. This output and the returned statement from the method were equivalent as indicated by the test.

---

testReverseDigits

```
// Test a string of length 0
assertEquals("", HW2.reverseDigits(""));

// Test a string of length 1 with no digits
assertEquals("h", HW2.reverseDigits("h"));

// Test a longer length string with no digits
assertEquals("this is one test", HW2.reverseDigits("this is one test"));
```

- These three tests are for the test 0, 1, many for the string lengths. These are important though, as they guarantee that nothing will be changed in the string if there are no real digits in the input. This is successfully observed in the tests, as strings of length 0, 1, and more than 1 are not impacted at all when being used as arguments in this method.

```
// Test a string with one digit
assertEquals("1", HW2.reverseDigits("1"));

// Test a string with numerically ordered digits
assertEquals("321", HW2.reverseDigits("123"));

// Test a string with disordered digits
assertEquals("145849", HW2.reverseDigits("948541"));
```

- These three tests test 1 and a couple variations of the many branch of the testing. Since there are only digits in these strings, I can accurately check whether or not digits are being properly flipped. In the string with only one digit, nothing is changed which is expected. For the string that counts 321, the method reverses it to count up as 123. For the final test of these three, the digits are placed in reverse order despite not being strictly increasing or decreasing. The fact

that all of these expected values align with the method output are revealed by the fact that these tests ran successfully.

```
// Test a string with many digits mixed into a few words
assertEquals("th3s w5rk0 1h", HW2.reverseDigits("th1s w0rk5 3h"));

// Test the example from the instructions
assertEquals("9 the d8gits of the4 string3 2 are1 reversed 0!",
        HW2.reverseDigits("0 the d1gits of the2 string3 4 are8 reversed 9!"));
```

- These two tests are very similar in that they mix in digits in with regular characters. I expected the same order of regular characters, with the digits being in the complete reverse order. This was confirmed by the assertions above, and the fact that example from the instructions works as intended proves that this method works.

---

testReplaceText

```
// Test strings of length 0
assertEquals("", HW2.replaceText("", ""));

// Test strings of length 1 with no replacements
assertEquals("h", HW2.replaceText("h", "h"));

// Test strings of length 5 with no replacements
assertEquals("hello", HW2.replaceText("hello", "hello"));
```

- These three tests test the 0, 1, many requirement for string without indicated substrings. Since nothing was indicated by parentheses, the output should be identical to the base input. This is observed and confirmed by the tests above.

```
// Test strings of length 2 indicated with only the replacements
assertEquals("", HW2.replaceText("()", "()"));

// Test strings of length one with indicated replacements
assertEquals("e", HW2.replaceText("(a)", "(e)"));
```

- Since I will be testing the 'many' branch for the rest of this JUnit section, these two tests are for the 0 and 1 part of the tests. Since I can't indicate a replacement when there is an empty string, I'm calling '()' an empty string. For the first test, an empty string is being replaced by another empty string, which should mean that the result is a true empty string devoid of parentheses. The second test here replaces a single letter. These two tests ran without output indicated a proper parent method.

```
// Test a base with no indicated substrings, but with ones indicated in
replacements
assertEquals("this is a test", HW2.replaceText("this is a test", "(this)"));

// Test a base with indicated substrings but none indicated in replacements
assertEquals("this  test", HW2.replaceText("this (is a) test", "nope"));
```

- These two tests check the behavior when there are indicated substrings in one of the inputs and not the other. When there are ones indicated in the replacements, they should be ignored and the base string should be returned unchanged. If there are indicated substrings in the base string but none in the replacements, they should be dropped. Because of the dropped substring in the base, there are two spaces instead of one because my method is designed to only impact substrings, not anything else around them. These two behaviors are exhibited by my written method as confirmed by the tests outlined above.

```
// Test strings with 1 substring in the base and 2 substrings in the
replacements
assertEquals("hello", HW2.replaceText("(goodbye)", "(hello) (world)"));

// Test strings with 2 substrings in the base and 1 substring in the
replacements
assertEquals("hello ", HW2.replaceText("(goodbye) (world)", "(hello)"));
```

- These are similar to the previous tests in that I am seeing what happens when there is not equal amounts of indicated substrings in the base and the replacements. For the first test I made sure to test what happens when there's one replacement that can be done but two are indicated. The first replacement should happen as expected, ignoring the second replacement indicated. For the second test, the base string has too many substrings indicated, so the replacement of the first one is done as planned but the second substring is dropped

completely. Since the two assertions ran without output, I guaranteed my code works as expected.

```
// Test strings with equal amounts of substrings in the two strings, but more
than 1 word per substring
assertEquals("this wasn't the first test", HW2.replaceText("this (is a) test",
"(wasn't the first)"));

// Test strings with equal amounts of substrings in the two strings, but more
than 1 substring per input
assertEquals("hello world", HW2.replaceText("(goodbye) (globe)", "(hello)
(world)"));
```

- These two tests ensure that the correct behavior is exhibited when the user does not input any mismatched parentheses. The first test, as indicated, uses multiple words in the substrings to ensure that the whole indicated phrase is replaced. The second test has multiple substrings indicated in equal magnitude to ensure that, when there is the right amount of substrings, all the replacements happen as expected. The assert statements here ran without throwing an error, indicating proper code.

```
/* Test strings with equal amounts of substrings in the two strings, but with
nested matched parentheses in the replacements
 * This is identical to the previous test but along with nested matched
parentheses */
assertEquals("hell()o wor()ld", HW2.replaceText("(goodbye) (globe)", "(hell()o)
(wor()ld)"));

/* Test strings with equal amounts of substrings in the two strings, but with
nested matched parentheses in the base
 * This is identical to the previous test but along with nested matched
parentheses */
assertEquals("hello world", HW2.replaceText("(good()bye) (glo()be)", "(hello)
(world)"));
```

- The instructions indicate that nested and matched parentheses should be treated as regular characters, which is what these are testing. The first test here has nested matching parentheses in the replacements, which should end up being in the output after replacements are done as

per the project criteria. The second test here has nested parentheses in the base string. These should be replaced along with the other characters in the substring as the nested parentheses are treated as regular characters as per the projects instructions. Both of these tests ran without error, indicated a correctly written method.

```
// Test strings with mismatched parentheses in the base but matched in the
replacements
assertEquals("this tests (a mismatched in the base",
        HW2.replaceText("this tests (a () in the base", "(mismatched)"));

// Test strings with matched parentheses in the base but unmatched in the
replacements
assertEquals("this tests a thing in the replacements",
        HW2.replaceText("this tests (mismatched parentheses) in the
replacements", "(a thing))"));
```

- This tests the behavior associated with mismatched parentheses. For the first test here, I mismatched parentheses in the base string, using a () to indicate where I want the actual replacement to go. The rogue '(' should remain in the output because of the behavior I outlined in the documentation of this method. The mismatched parentheses should be ignored, meaning that they are not considered in the replacement. Keeping this in mind, the second test here also works as intended except with the mismatch in the replacements. Since the mismatched parentheses are ignored, they are not included in the output and 'a thing' is the only part of the replacements that shows up after calling the method. Both of these tests ran as expected, indicating my code and reasoning works here.

```
// Test mismatched parentheses in the base with mismatched parentheses in the
replacements
assertEquals("testing mismatches)", HW2.replaceText("testing (mis()matches))",
"((mismatches)"));

// Test a base with too many substrings that are also  mismatched, replacements
are not mismatched
assertEquals("hello (globworld ", HW2.replaceText("hello (glob(e) (nope)", "
(world)"));
```

- These tests expound upon the previous ones in that they test more mismatched edge cases. In the first test here, there are mismatches in both the base and the replacements. As explained earlier, any mismatches should be ignored completely in determining what to and what not to replace. For this test, the closing bracket at the end of the base is left in the output, but the opening parenthesis at the start of the replacements are dropped.
- In the second test, I made sure to have an abundance of substrings in the base input, while also making them have mismatched parentheses. In this case, where the replacements are not mismatched, '(e)' should be replaced with world and '(nope)' should be dropped while including the space afterwards. This is observed in the assertions, and both of these tests ran as expected.

```
// Test a base with correct everything but replacements with too many
substrings that are also mismatched
assertEquals("this is a trial of robustness",
        HW2.replaceText("this is a (test) of robust()", "(trial)) (ness) (do
not include)"));

// Test a base with mismatched parentheses with too many substrings for
mismatched replacements
assertEquals("this is a trial of robustness from  me",
        HW2.replaceText("this is a (test) of robust() (do not include) me", "
((trial) of (ness from)"));
```

- These two tests further develop the tests explained in the previous bullet point. For the first test here, I made sure the base string was matched in terms of its parentheses, while making the replacements unmatched. There is also one too many substrings in the replacements, and they are ignored as a result. '(trial)' and '(ness)' are the only ones considered and implemented, as indicated by my expected string.
- For the second test, I had too many substrings indicated in the base string for the amount indicated in the replacements. Here, '(do not include)' is dropped from the output, and '(trial)' and '(ness from)' are placed where their respective substrings in the base are. This is exhibited in my expected argument for assert, and the running of the test indicates correct code.

```
// Test a base with mismatched parentheses with not enough substrings for
matched replacements
assertEquals("this is (a TRIAL of robustness from me",
        HW2.replaceText("this is (a (test) of robust() me", "(TRIAL) (ness
```

```
from) (please don't include me)"));

// Test a base with mismatched parentheses with too many substrings for matched
replacements
assertEquals("this is (a TRIAL of robustness from me ",
        HW2.replaceText("this is (a (test) of robust() me (please don't include
me)", "(TRIAL) (ness from)"));
```

- This is the final 'stress-test' of this method. First, I tested what happens when there are mismatched parentheses in the base in conjunction with there being too many substrings in the replacements. My expected string contains the correct replacements of the substrings, with the mismatched parenthesis still staying in the output. '(please don't include me)' is correctly omitted. This behavior is confirmed by the running of this test with the outlined arguments.
- For the second test here, it is almost identical in set-up, only differing in the fact that the '(please don't include me)' substring is in the base this time. Here, the expected behavior should be that the mismatched parentheses is observed in the output, the substring indicated in the previous sentence is ignored, and the two proper replacements occur correctly. This behavior is exhibited correctly as proved by the running of these assertions.

```
// Test the first example from the instructions
assertEquals("a cool programming problem",
        HW2.replaceText("a (simple) programming (example)", "(cool)
(problem)"));

// Test the second example from the instructions
assertEquals("a answer with really (two) not three replacements ",
        HW2.replaceText("a ((nested) example) with (three) replacements (to
(handle))", "the replacements are (answer) and (really (two) not three)"));
```

- Despite all of the complex tests above, these are straight from the instructions for this project, as indicated. These tests were included to ensure I didn't break anything in the basic workings of this method when trying to properly cover edge cases. The lack of error thrown when running these tests indicates correct code.

---

testReverseAll

About this method

I unfortunately was unable to get to the coding of this method.

> ***That concludes the testing portion of this project***
>
> ```
> I appologize for funky formatting and length
> ```