proj5/BSTNode.java

```java
package proj5;

/** A reusable node class, since it can hold E data.
 *
 * @author Chris Fernandes and Advitya Singh
 * @version June 6, 2025
 */
public class BSTNode<E> {

    // 3 instance variables
    // data is the object that the BST is holding
    // llink is the left node of the this node
    // rlink is the right node of this node
    public E data;
    public BSTNode<E> llink;
    public BSTNode<E> rlink;

    /**
     * non-default constructor
     * @param newKey string that node will hold
     */
    public BSTNode(E newKey)
    {
        data = newKey;
        llink = null;
        rlink = null;
    }

    /**
     * returns key as printable string
     */
    public String toString()
    {
        return (String) data;
    }
}

```

proj5/BinarySearchTree.java

```java
/**
 * Binary Search Tree Data Structure. Allows user to hold any comparable data type. Allows BST operations like searching, inserting, deleting, etc.
 * @author: Advitya Singh and Chris Fernandez
 * @version June 6, 2025
 *
 * Honor Code: I affirm that I have carried out the attached academic endeavors with full academic honesty, in
 * accordance with the Union College Honor Code and the course syllabus.
 */

package proj5;

public class BinarySearchTree<E extends Comparable<E>>
{
    // Only one instance variable: root of BST
    private BSTNode<E> root;

    /**
     * Default constructor: creates BST.
     */
    public BinarySearchTree() {
        root = null;
    }

    /**
    * inserts recursively.  I include this one so you can
    * make your own trees in the testing class
    *
    * @param subroot inserts into subtree rooted at subroot
    * @param newNode node to insert
    * @return the BST rooted at subroot that has newNode inserted
    */
    private BSTNode<E> recursiveInsert(BSTNode<E> subroot, E comparableItem) {
        if (subroot == null) {
            return new BSTNode<>(comparableItem);
        }
        else if (comparableItem.compareTo(subroot.data) > 0) {
            subroot.rlink = recursiveInsert(subroot.rlink, comparableItem);
            return subroot;
        }
```

```
        else {  // newNode.data smaller than subroot.data, so newNode goes on left
            subroot.llink = recursiveInsert(subroot.llink, comparableItem);
            return subroot;
        }
    }

    /**
    * inserts recursively. Use this in your JUnit tests to
    * build a starting tree correctly
    *
    * @param newString String to insert
    */
    public void insert(E ComparableItem){
        root = recursiveInsert(root, ComparableItem);
    }

    /**
     * Helper method for the search method.
     * @param current Node currently checking if target is at.
     * @param target String to look for
     * @return boolean true is string is in node, false if not.
     */
    private boolean recursiveSearch(BSTNode<E> current, E target) {
        if (current == null) {
            return false;
        }

        if (target.compareTo(current.data) == 0 ){
            return true;
        }

        else if (target.compareTo(current.data) > 0) {
            return recursiveSearch(current.rlink, target);
        }
        else {
            return recursiveSearch(current.llink, target);
        }
    }

    /**
     * Checks if a certain string is in the BST recursively.
     * @param target String the user is looking for.
     * @return boolean true is target in BST, false if not.
     */
```

```
84      public boolean search(E target) {
85          return recursiveSearch(root, target);
86      }
87
88      /**
89       * Deletes a value from the BST.
90       * @param value to be deleted from the BST.
91       */
92      public void delete(E value) {
93          root = delete(root, value);
94      }
95
96      /**
97       * Private helper method for delete and does the recursion.
98       * @param subroot
99       * @param value
100      * @return
101      */
102     private BSTNode<E> delete(BSTNode<E> subroot, E value) {
103         // Case: value is not in tree.
104         if (subroot == null) {
105             return null;
106         }
107
108         int comparator = value.compareTo(subroot.data);
109
110         // 3 Cases:
111         if (comparator < 0) {
112             subroot.llink = delete(subroot.llink, value);
113         } else if (comparator > 0) {
114             subroot.rlink = delete(subroot.rlink, value);
115         } else {
116             // Value found: 4 deletion cases
117             // Case 1: No children (leaf)
118             if (subroot.llink == null && subroot.rlink == null) {
119                 return null;
120             }
121
122             // Case 2: Only right child
123             else if (subroot.llink == null) {
124                 return subroot.rlink;
125             }
126
127             // Case 3: Only left child
128             else if (subroot.rlink == null) {
```

```
129                 return subroot.llink;
130             }
131 
132             // Case 4: Two children
133             else {
134                 // Find in-order successor (leftmost node of right subtree)
135                 BSTNode<E> successor = findMin(subroot.rlink);
136 
137                 // Copy successor's data into current node
138                 subroot.data = successor.data;
139 
140                 // Delete successor from right subtree
141                 subroot.rlink = delete(subroot.rlink, successor.data);
142             }
143         }
144 
145         return subroot;
146     }
147 
148     /**
149      * Helper method for recursiveDelete. Finds the minimum value in the
    tree.
150      * @param node finds the node with the smallest value.
151      * @return
152      */
153     private BSTNode<E> findMin(BSTNode<E> node) {
154         while (node.llink != null) {
155             node = node.llink;
156         }
157         return node;
158     }
159 
160 
161     /**
162      * Method to find a return a comparable object present in the BST.
163      * @param target object to look for and return.
164      * @return The object looking for, if found. Else, null.
165      */
166     public E find(E target) {
167         return findHelper(root, target);
168     }
169 
170     /**
171      * Recursive helper method for find.
172      * @param node to look through.
```

```
173      * @param target object to look for.
174      * @return E object when found, or else null.
175      */
176     private E findHelper(BSTNode<E> node, E target) {
177         if (node == null) return null;
178 
179         int comparator = target.compareTo(node.data);
180         if (comparator == 0) return node.data;
181         else if (comparator < 0) return findHelper(node.llink, target);
182         else return findHelper(node.rlink, target);
183     }
184 
185     /**
186      * Returns the number of data items (nodes) in the tree.
187      * @return int for number of nodes
188      */
189     public int size() {
190         return recursiveSize(root);
191     }
192 
193     /**
194      * Recursive helper method for size()
195      * @param node Node to be accounted for in final size
196      * @return number of nodes so far
197      */
198     private int recursiveSize(BSTNode<E> node) {
199         if (node == null) {
200             return 0;
201         }
202 
203         return 1 + recursiveSize(node.llink) + recursiveSize(node.rlink);
204     }
205 
206     /**
207      * Returns a String showing the contents of the tree using inorder
    traversal.
208      */
209     public String toString() {
210         return toString(root);
211     }
212 
213     /**
214      * Recursive helper method for toString method.
215      * @param node to be extracted string from.
216      * @return String of the BST so far.
```

```
     */
    private String toString(BSTNode<E> node) {
        if (node == null) {
            return "";
        }

        String left = toString(node.llink);
        // \n is put in here but could be removed and put into Thesaurus and
WordCounter.
        String middle = (node.data).toString() + "\n";
        String right = toString(node.rlink);
        return left + middle + right;
    }


    /** recursive helper for toStringParen
     *
     * @param subroot root of subtree to start at
     * @return inorder string of elements in this subtree
     */
    private String toStringParen(BSTNode<E> subroot) {
      if (subroot == null) // base case
          return "";
      else
          return "(" + toStringParen(subroot.llink) + " " +
                  subroot.toString() + " " + toStringParen(subroot.rlink) +
")";
    }

    /**
     * returns string showing tree structure using parentheses, as shown in
class
     */
    public String toStringParen() {
      return toStringParen(root);
    }

}
```

proj5/EntryForThesaurus.java

```java
/**
 * One possible Comparable data entry for a BST Node.
 * Thesaurus inserts objects of type EntryForThesaurus into BST.
 * @author: Advitya Singh
 * @version: June 6, 2025
 */

package proj5;

public class EntryForThesaurus implements Comparable<EntryForThesaurus> {

    // two instance variables: word itself and and array of its synonyms
    private String word;
    private String[] synonyms;

    /**
     * Non-default constructor. Constructs a EntryForThesaurus object from given word and synonym list.
     * @param word the word itself
     * @param synonymArray its synonyms
     */
    public EntryForThesaurus(String word, String[] synonymArray) {
        this.word = word;
        synonyms = synonymArray;
    }

    /**
     * Default constructor. Constructs a EntryForThesaurus object from given word
     * and initializes array to an empty array.
     * @param word
     */
    public EntryForThesaurus(String word) {
        this.word = word;
        synonyms = new String[0];
    }

    /**
     * getter for word
     * @return word itself
     */
    public String getWord() {
        return this.word;
```

```
    }

    /**
     * getter for the synonym array
     * @return the synonym array
     */
    public String[] getSynonyms() {
        return this.synonyms;
    }

    /**
     * Adds synonyms if the word already exists in the array
     * @param toAdd the array of words to add
     */
    public void addSynonyms(String[] toAdd){
        int uniqueCount = 0;
        for (String newWord : toAdd) {
            boolean isDuplicate = false;
            for (String existing : synonyms) {
                if (existing.equals(newWord)) {
                    isDuplicate = true;
                    break;
                }
            }
            if (!isDuplicate) {
                uniqueCount++;
            }
        }

        // Create a new array with the correct new size
        String[] combined = new String[synonyms.length + uniqueCount];
        int index = 0;

        // Copy over existing synonyms
        for (String s : synonyms) {
            combined[index++] = s;
        }

        // Add new unique ones
        for (String newWord : toAdd) {
            boolean isDuplicate = false;
            for (String existing : synonyms) {
                if (existing.equals(newWord)) {
                    isDuplicate = true;
                    break;
```

```
                }
            }
            if (!isDuplicate) {
                combined[index++] = newWord;
            }
        }

        // Update the field
        synonyms = combined;
    }

    /**
     * compareTo method that compares two EntryForThesaurus object.
     */
    @Override
    public int compareTo(EntryForThesaurus other) {
        return this.word.compareToIgnoreCase(other.word);
    }

    /**
     * toString public method for an EntryForThesaurus object.
     * For example: really - {syn1, syn2,..., synn}
     */
    @Override
    public String toString() {
        return word.toLowerCase() + " - " + synonymsToString();
    }

    /**
     * helper method for EntryForThesaurus toString() method.
     * @return a string of synonyms like {syn1, syn2,..., syn3}
     */
    private String synonymsToString(){
        if (synonyms.length == 0) return "{}";

        String result = "{";
        for (int i = 0; i < synonyms.length; i++) {
            result += synonyms[i];
            if (i < synonyms.length - 1) result += ", ";
        }
        result += "}";
        return result;
    }
}

```

proj5/Thesaurus.java

```java
/**
 * Data structure that holds words and their associated synonyms. You can look up a word and retrieve a synonym for it.
 * @author: Advitya Singh and Chris Fernandez
 * @version: June 6, 2025
 */

package proj5;
import java.util.Random;

public class Thesaurus<E> {

    // Only one instance variable: a BST holding EntryForThesaurus object as data.
    private BinarySearchTree<EntryForThesaurus> BSTThesaurus;

    /**
     * Default constructor. Creates an empty thesaurus.
     */
    public Thesaurus(){
        BSTThesaurus = new BinarySearchTree<>();
    }

    /**
     * Builds a thesaurus from a text file.
     * Each line of the text file is a comma-separated list of synonymous words. The first word
     * in each line should be the thesaurus entry. The remaining words
     * on that line are the list of synonyms for the entry.
     *
     * @param file path to comma-delimited text file
     */
    public Thesaurus(String file){
        BSTThesaurus = new BinarySearchTree<>();

        LineReader reader = new LineReader(file, ",");
        String[] eachLine;

        while ((eachLine = reader.getNextLine()) != null) {
            if (eachLine.length > 1) {
                String firstWord = eachLine[0].toLowerCase();
                String[] synonymList = new String[eachLine.length - 1];
                for (int i = 1; i < eachLine.length; i++) {
```

```
                    synonymList[i - 1] = eachLine[i];
                }
                insert(firstWord, synonymList);
            }

            else if (eachLine.length == 1){
                String firstWord = eachLine[0].toLowerCase();
                String[] synonymList = {};
                insert(firstWord, synonymList);
            }
        }
        reader.close();
    }

    /**
     * removes entry (and its associated synonym list) from this thesaurus.
     * If entry does not exist, do nothing.
     *
     * @param entry word to remove
     */
    public void delete(String entry){
        EntryForThesaurus entryToDelete = new EntryForThesaurus(entry);
        BSTThesaurus.delete(entryToDelete);
    }

    /**
     * Gets a random synonym for the given keyword.
     * If keyword does not exist, return the empty string.
     *
     * @param keyword word to find a synonym for
     * @return a random synonym from the synonym list of that word,
     * or empty string if keyword doesn't exist.
     */
    public String getSynonymFor(String keyword){
        EntryForThesaurus ThesaurusInput = new EntryForThesaurus(keyword);
        if (BSTThesaurus.search(ThesaurusInput) == false){
            return "";
        }

        else {
            Random rand = new Random();
            String[] synonymList =
BSTThesaurus.find(ThesaurusInput).getSynonyms();
            if (synonymList.length == 0) {
                return "";
```

```
            }
            int randomIndex = rand.nextInt(synonymList.length);
            return synonymList[randomIndex];
        }
    }

    /**
     * inserts entry and synonyms into thesaurus.
     * If entry does not exist, it creates one. If it does exist, it adds the given
     * synonyms to the entry's synonym list.
     *
     * @param entry keyword to be added
     * @param syns array of synonyms for keyword entry
     */
    public void insert(String entry, String[] syns){
        EntryForThesaurus temp = new EntryForThesaurus(entry.toLowerCase());

        if (BSTThesaurus.search(temp)){
            BSTThesaurus.find(temp).addSynonyms(syns);
        }

        else {
            BSTThesaurus.insert(new EntryForThesaurus(entry, syns));
        }
    }

    /**
     * return this thesaurus as a printable string. Each keyword
     * and synonym list should be on its own line. The format of
     * each line is: <keyword> - {<syn1>, <syn2>, ..., <synN>}
     * For example, happy - {glad, content, joyful}
     * jump - {leap, bound}
     * The thesaurus keywords will be in alphabetical order.
     * The order of the synonym list words is arbitrary.
     *
     * @return String representation
     */
    public String toString() {
        return BSTThesaurus.toString();
    }
}

```

proj5/ThesaurusTester.java

```java
/**
 * /**
 * Testing file for Thesaurus Object.
 * @author: Advitya Singh
 * @version: June 6, 2025
 */
package proj5;
import org.junit.*;

import static org.junit.Assert.*;
import org.junit.rules.Timeout;

public class ThesaurusTester {

    @Rule
    public Timeout timeout = Timeout.millis(100);

    // testing default constructor using toString. should be empty string
    @Test
    public void testConstructor() {
        Thesaurus<EntryForThesaurus> thesaurus = new Thesaurus<>();

        assertEquals(thesaurus.toString(), "");
    }

    // Inserting one word and synonyms into Thesaurus
    @Test
    public void testInsertAndToStringSingle() {
        Thesaurus<EntryForThesaurus> thesaurus = new Thesaurus<>();
        thesaurus.insert("happy", new String[]{"joyful", "cheerful"});

        String output = thesaurus.toString();
        assertTrue(output.contains("happy - {joyful, cheerful}"));
    }

    // testing inserting into thesaurus more than once
    @Test
    public void testAddSynonymsToExistingWord() {
        Thesaurus<EntryForThesaurus> thesaurus = new Thesaurus<>();
        thesaurus.insert("smart", new String[]{"intelligent"});
        thesaurus.insert("smart", new String[]{"clever", "intelligent"}); //
"intelligent" duplicate

```

```
43          String output = thesaurus.toString();
44          assertTrue(output.contains("smart - {intelligent, clever}") ||
45                  output.contains("smart - {clever, intelligent}"));
46      }
47
48      // inserting and deleting from thesaurus
49      @Test
50      public void testDeleteRemovesWord() {
51          Thesaurus<EntryForThesaurus> thesaurus = new Thesaurus<>();
52          thesaurus.insert("cold", new String[]{"chilly", "freezing"});
53          thesaurus.delete("cold");
54
55          String output = thesaurus.toString();
56          assertFalse(output.contains("cold"));
57      }
58
59      // trying to delete a non-existing word.
60      @Test
61      public void testDeleteNonExisting() {
62          Thesaurus<EntryForThesaurus> thesaurus = new Thesaurus<>();
63          thesaurus.delete("cold");
64
65          String output = thesaurus.toString();
66          assertFalse(output.contains("cold"));
67      }
68
69
70      // testing getSynonym for a word with two synonyms
71      @Test
72      public void testGetSynonymForExistingWord() {
73          Thesaurus<EntryForThesaurus> thesaurus = new Thesaurus<>();
74          thesaurus.insert("bright", new String[]{"shiny", "luminous"});
75          String synonym = thesaurus.getSynonymFor("bright");
76
77          assertTrue(synonym.equals("shiny") || synonym.equals("luminous"));
78      }
79
80      // getting synonyms for a non-existing word should be am empty string.
81      @Test
82      public void testGetSynonymForNonexistentWord() {
83          Thesaurus<EntryForThesaurus> thesaurus = new Thesaurus<>();
84          String synonym = thesaurus.getSynonymFor("nonexistent");
85
86          assertEquals("", synonym);
87      }
```

```
88
89      // further testing for the thesaurus was done by printing to the console
    using smallThesaurus.txt.
90
91  }
92
```

proj5/WCounterInput.java

```java
/**
 * Comparable data entry for a BST Node.
 * WordCounter inserts objects of type WCounterInput into BST.
 * @author: Advitya Singh
 * @version: June 6, 2025
 */

package proj5;

public class WCounterInput implements Comparable<WCounterInput> {

    // two instance variables, the word itslef and the wordFrequency is the number of times it occurs
    private String wordItself;
    private int wordFrequency;

    /**
     * Constructor: constructs a word with 1 frequency.
     * @param word
     */
    public WCounterInput(String word){
        wordItself = word;
        wordFrequency = 1;
    }

    /**
     * increases frequency by 1.
     */
    public void increaseFrequency(){
        wordFrequency++;
    }

    /**
     * returns the word itself
     * @return the String of the word
     */
    public String getWord(){
        return wordItself;
    }

    /**
     * returns the number of times the word occurs
     * @return the int frequency instance variable
```

```
43      */
44     public int getFrequency(){
45         return wordFrequency;
46     }
47 
48     /**
49      * compareTo for a WCounterInput object.
50      * Uses the simple word compare to that compares alphabetically.
51      */
52     @Override
53     public int compareTo(WCounterInput other) {
54         return this.wordItself.compareTo(other.getWord());
55     }
56 
57     /**
58      * toString for a WCounterInput object.
59      * For example: hello: 2
60      */
61     public String toString() {
62         return getWord() + ": " + getFrequency();
63     }
64 
65 }
66 
```

proj5/WordCounter.java

```java
/**
 * An ADT for computing word frequencies from a text file.
 * @author: Advitya Singh and Chris Fernandez
 * @version: June 6, 2025
 */

package proj5;

public class WordCounter {

    // Only one instance variable: a BST holding WCounterInput object as data.
    private BinarySearchTree<WCounterInput> BSTWordCounter;

    /*
     * Default constructor: creates a BST for the WordCounter.
     */
    public WordCounter(){
        BSTWordCounter = new BinarySearchTree<>();
    }

    /**
     * Computes frequency of each word in given file.
     * @param file path to file, such as "src/input.txt".
     */
    public void findFrequencies(String file){
        LineReader reader = new LineReader(file, " ");
        String[] words;

        while ((words = reader.getNextLine()) != null) {
            for (String rawWord : words) {
                String cleanedWord = rawWord.replaceAll("[^a-zA-Z]", "").toLowerCase();
                if (!cleanedWord.isEmpty()) {
                    insertOrIncreaseFrequencyinBSTWordCounter(cleanedWord);
                }
            }
        }
        reader.close();
    }

    /**
```

```
41      * inserts a word into WordCounter or increases frequency if the word
    already exists
42      * @param rawWord word to insert into WordCounter
43      */
44     private void insertOrIncreaseFrequencyinBSTWordCounter(String rawWord){
45         WCounterInput currentWord = new WCounterInput(rawWord);
46 
47         if (BSTWordCounter.search(currentWord) == false){
48             BSTWordCounter.insert(currentWord);
49         }
50 
51         else {
52             WCounterInput existing = BSTWordCounter.find(currentWord);
53             existing.increaseFrequency();
54         }
55     }
56 
57     /**
58      * returns the frequency of the given word
59      * @param word string to get the frequency of
60      * @return the number of times word appears in the input file
61      */
62     public int getFrequency(String word){
63         word = word.replaceAll("[^a-zA-Z]", "").toLowerCase();
64         WCounterInput input = new WCounterInput(word);
65         WCounterInput InputNode = BSTWordCounter.find(input);
66         if (InputNode == null) return 0;
67         return InputNode.getFrequency();
68     }
69 
70     /**
71      * returns words and their frequencies as a printable String.
72      * Each word/frequency pair should be on a separate line, and
73      * the format of each line should be <word>: <frequency>.
74      * For example,
75      * are: 3
76      * bacon: 2
77      * Words should be in alphabetical order.
78      * @return string form of wordcounter
79      */
80     public String toString(){
81         return BSTWordCounter.toString().trim();
82     }
83 }
84 
```

proj5/WordCounterTester.java

```java
/**
 * Testing file for WordCounter Object.
 * @author: Advitya Singh
 * @version: June 6, 2025
 */
package proj5;

import org.junit.*;

import static org.junit.Assert.*;
import org.junit.rules.Timeout;


public class WordCounterTester {

    private WordCounter wordCounter;

    @Rule
    public Timeout timeout = Timeout.millis(100);

    @Before
    public void setUp() {
        wordCounter = new WordCounter();
    }


    // Constructing a wordCounter testing using toString
    @Test
    public void testConstructor(){
        assertEquals(wordCounter.toString(), "");
    }

    // from this point, all testing for wordCounter is done using the apartment text provided.

    // testing findFrequency and getFrequency: word occuring once
    @Test
    public void findingFrequencyForWordOccuringOnce(){
        wordCounter.findFrequencies("proj5/apartment.txt");
        assertEquals(wordCounter.getFrequency("when"), 1);
    }

    // testing findFrequency and getFrequency: word occuring five times
```

```
43      @Test
44      public void getFrequencyForWordOccuringMultiple(){
45          wordCounter.findFrequencies("proj5/apartment.txt");
46          assertEquals(wordCounter.getFrequency("grungy"), 5);
47      }
48
49      // testing findFrequency and getFrequency: lastWord
50      @Test
51      public void getFrequencyForLastWord(){
52          wordCounter.findFrequencies("proj5/apartment.txt");
53          assertEquals(wordCounter.getFrequency("insane"), 3);
54      }
55
56      // testing getFrequency for non-existing word
57      @Test
58      public void getFrequencyNonexistingWord(){
59          wordCounter.findFrequencies("proj5/apartment.txt");
60          assertEquals(wordCounter.getFrequency("yes"), 0);
61      }
62
63      // toString for entire wordCounter was done in main method by printing.
64  }
65
```

proj5/GrammarChecker.java

```java
/**
 * Uses a thesaurus and word frequencies to replace overused words in a text document with random synonyms.
 * @author: Advitya Singh and Chris Fernandez
 * @version: June 6, 2025
 */
package proj5;

public class GrammarChecker {

    int threshold;
    Thesaurus<EntryForThesaurus> thesaurus;

    /**
     * Non-default constructor. Builds a thesaurus out of the
     * given comma-separated file and sets the threshold for overused words.
     *
     * @param thesaurusFile path to comma-separated file used to build a thesaurus
     * @param threshold a word is considered "overused" if it appears more than
     * (but not equal to) this many times in a text document
     */
    public GrammarChecker(String thesaurusFile, int threshold){
        thesaurus = new Thesaurus<>(thesaurusFile);
        this.threshold = threshold;
    }

    /**
     * Given a text file, replaces overused words with synonyms.
     * Finished text is printed to the console.
     *
     * @param textfile file with original text
     */
    public void improveGrammar(String textfile) {
        WordCounter wordCounter = new WordCounter();
        wordCounter.findFrequencies(textfile);

        LineReader reader = new LineReader(textfile, " ");

        String[] line;
        while ((line = reader.getNextLine()) != null) {

```

```
            for (int i = 0; i < line.length; i++) {
                String word = line[i];

                String cleanWord = word.replaceAll("[^a-zA-Z]",
"").toLowerCase();
                String punctuation = word.replaceAll("[a-zA-Z]", "");
                int frequency = wordCounter.getFrequency(cleanWord);

                if (frequency > threshold) {
                    String synonym = thesaurus.getSynonymFor(cleanWord);
                    if (!synonym.equals("")) {
                        String replaced =
matchCapitalization(word.replaceAll("[^a-zA-Z]", ""), synonym);
                        System.out.print(replaced + punctuation + " ");
                    } else {
                        System.out.print(word + " ");
                    }
                } else {
                    System.out.print(word + " ");
                }
            }
        }
        }

    /**
     * Helper method for improveGrammar. Helps make sure two words of
different capitalization count as the same word.
     * @param original original string encountered in text.
     * @param replacement what to replace the non-word charachters like
punctuation with. Usually an empty string.
     * @return lowercase of the original word without punctuation.
     */
    private String matchCapitalization(String original, String replacement) {
        if (original.toUpperCase().equals(original)) {
            return replacement.toUpperCase();
        } else if (Character.isUpperCase(original.charAt(0))) {
            return Character.toUpperCase(replacement.charAt(0)) +
replacement.substring(1);
        } else {
            return replacement.toLowerCase();
        }
    }

}

```