For this project, you will write a program that creates a Markov chain from input text files and generates gibberish that ends up adhering to many of the basic grammar rules of the language. When the order is set to the maximum level (3), the generated text will not sound like a random string of words but like actual sentences where many seem to even have meaning. However, when a few generated sentences are strung together it becomes clear that the text is gibberish.

Your program must have the following classes:

**Markov.java**: to contain the main method for the program. The Markov program must take at least 2 command-line arguments: The order of the Markov chain and the number of sentences to be generated in the output.

This class must build the Markov chain using all .txt files in the data directory.

Must read the following “words”:

1) Any sequences of alphabetic characters, including the apostrophe character, delimited by one or more spaces, commas, periods or linefeeds.
2) The three punctuation marks recognized by your program (period, question mark and comma) are each counted as a word.
3) Whenever an explanation point character, !, is read, change it to a period.
4) Ignore any sentence that contain only one word (other than a period, ! or ?). This is actually a bit tricky to do because it means you need to read multiple words possibly skipping all caps words, before you can add a word to the data structure. Therefore, we will skip this.
5) Ignore any punctuation mark (period, comma or question mark) that follows a punctuation mark. This is easy to implement: in addWord(word), if word is a parent punctuation mark and its parent is also a punctuation mark, then return without doing anything.
6) Ignore any word that has more than one letter and all letters are uppercase.
7) If the hyphen ‘-‘ character occurs at the end of a line, then do not count that newline character ‘/n’ as making ending the word. Otherwise, any number of spaces or newline characters ends the current word.
8) All other characters (numbers, special symbols such as ‘:’ ‘$’ ‘@’ ‘;' ‘l’ ‘(’ etc) must NOT be included in any words (just ignore these characters).

9) If a word is read that that starts with a lowercase letter AND that same word is already in the map starting with an uppercase letter, then replace the word starting with the uppercase letter with the lower case version. Similarly, if a word starting with an uppercase letter is read and that word is already in the map as starting with a lowercase letter, then do not add the uppercase version to the map.

10) The generated text:
    a. Must be printed to the console.
    b. Must contain the number of sentences given in the input.
    c. Must be generated as defined using the order, TreeMaps and probabilities.

**Node.java**: This is a simple data class. There are no methods and all fields are public. **Node** must have the fields:

a. **public String word;** When adding a Node to a **TreeMap**, **node.word** is the key.

b. **public java.util.TreeMap <String, Node> children = null;**
   Note: The top level **TreeMap** is in **WordChain**. Each value in that **TreeMap** is one of these **Node** instances. Each of these **Node** instances has its own **TreeMap** and each value in that **TreeMap** is another instance of **Node** which has yet a deeper **TreeMap**. This is how the Markov chain structure is build.

c. **public int count = 1;** This field must be maintained so that it equals the total number of occurrences of this word with the ancestry defined by the node’s level in **WordChain**. For example, for some input text, he word “car” appears 25 times. Where 15, of those times it comes occurs after the word “the”, 7 times after the word “a“ and 3 times after the word “my”. In this example, the word “car” would be in 4 nodes of the total Markov chain: 1 at the top level with count = 25, and 3 at the second level where “car” is a child of “the”, where a different “car” node is a child of “a” and where the final “car” node is a child of “my”.

   Note: in the **TreeMap** data structure, there may NOT be ANY REPEATED KEYS. In the above example, the word “car” is used as a key 4 different times; however, each time, it is a key in a DIFFERENT **TreeMap**.

**WordChain.java**: This represents the Markov Chain. **WordChain.java** must implement the following fields and methods:
a. `private java.util.TreeMap<String, Node> chain;` This will be the root level chain. It is searched when matching a chain at the ancestry level equal to order. For example, if order=2, parent and you are looking for a word

b. `private static final Node FOSTER_PARENT = new Node();`

c. `public static final int MAX_ORDER = 3;`

d. `public final int ORDER;` This Any field declared final, can never be changed and MUST be given a value EITHER when declared or in the constructor.

e. `private Node[] ancestry = new Node[MAX_ORDER];`

f. `private static final int PARENT = 0;`

g. `private static final int GRANDPARENT = 1;`

h. `private static final int GREATGRANDPARENT = 2;`

i. `public WordChain(int order) -- This constructor will take one integer argument, [0,MAX_ORDER], representing the order of the chain. If this constructor is called with a value of order out of range, then the constructor must throw an IllegalArgumentException. The constructor’s body must contain the code:

```
ORDER = order;
chain = new java.util.TreeMap();
chain.put(".", FOSTER_PARENT);
ancestry[PARENT] = FOSTER_PARENT;
ancestry[GRANDPARENT] = null;
ancestry[GREATGRANDPARENT] = null;
```

Why? What are each these lines doing?

j. `public void addWord(String word) – This method must add a word to the WordChain. When order=3, the class must use the ancestry array to keep track of the previous three words added (the given word’s parent, grandparent and great-grandparent). When order=2, the class must keep track of the previous two words. When order=1, the class need only keep track of the previous word. Finally, when order=0, all words have no known ancestry (all words are basters). Any word that follows a period, explanation point, question mark or is the first word in a file, must be added as though it’s parent is a period, ‘.’ and its grand ancestors are null. This is why the “.” note added at the base level of the tree is named “FOSTER_PARENT”.

If the word is not yet in chain, then this method must create a new Node node, (setting node.word to word, leaving its count = 1 and its children = null. Then it must add the (key=word, value=node) pair to chain.

If the word is already in chain, then get the word’s value (Node node), and increment node.count.
Then, if order is at least 1, get the node in the base level tree of the greatest non-null
ancestor and either add this word as a new child (creating a new Node and it adding to
the ancestor’s children) or increment the count of the child of that ancestor with word
equal to this word.

This process must be repeated for each non-null ancestor; however, each newer
generation uses one additional depth level of the TreeMaps.

Finally, at the end of this `addWord()` method, the current great-grandparent is
forgotten, the grandparent becomes the great-grandparent, the parent becomes the
grandparent and the child node of the current parent with key=`word` becomes the
new parent.

If the word added is a period ‘.’ or question mark ‘?’, then the parent must be set to
FOSTER_PARENT, and greater ancestors must be set to `null`.

k. `public String generateSentence(Random rand)` – Must generate and
return a sentence, using the provided random number generator and the completely
finished Markov chain. The Generated sentences must follow the following rules:

1) The first word of the sentence must be a child of ‘.’ at the base level.

2) Every word in the sentence must be chosen with probability that is directly
proportional to its count relative to its parent’s count. If order=1 or if the
grandparent is null, then the parent’s word is used as the key on the base level
TreeMap. If order=2 or great=grandparent is null, then the grandparent’s word is
used as a key on the base level and the parent’s word is used as a key on the
child TreeMap of the grandfather. If order=3, then the great-grandparent’s word
is used as a key on the base level, etc.

3) A sentence ends when a period or question mark is chosen as the next word.

4) A space must be added to the returned String before each word EXCEPT if the
word is a period, a question mark or a comma or if the word is the first word on
in the sentence.

5) Change the first character of the first word of each sentence to an uppercase
letter.

**Important Hints and Guidelines**

Regular expressions are very useful in splitting Strings; however, since we need to treat
commas and periods and question marks as words (not just consumable delimiters), the regular
expression you would need is rather complex. I think the simplest way is to read one character
at a time. Do this with the `read()` method of the `BufferedReader`. This method reads the
character as an int or returns -1 if the end of file is reached. Use it like this:

```
c = 0;
int ascii = reader.read();
if (ascii != -1) c = (char) ascii;
```
In the version I just wrote, I placed this in a method called String readWord() that does the following:

1) Start with String word=""; This will be the word returned.
2) Loop while word.length() is equal to zero.
3) If end-of-file is read when the length of word is zero, then return null. However, if word has a length greater than zero when end-of-file is read, then return word.
4) Start reading characters (and ignoring them), one at a time, until a non-whitespace character is read:

   while (Character.isWhitespace(c))
   
   Nothing needs to be done in that loop except to read the next character and break if end-of-file.
5) Once you have read a non-whitespace character, if is an alphabetic character or an apostrophe, then append it to word.
6) If the character is I change it to a period.
7) Keep reading characters until either you reach the end of the file, read a whitespace character or read a punctuation mark (period, comma or question mark). After exiting this inner loop, if word still has a length of 0, then the process restarts at step 2.
8) This readWord() method must somehow deal with the case of reading a word followed by punctuation mark. In this case, readWord() has actually reads two words (the word and the punctuation mark), but, as defined, only returns one word. I handled this by creating a String class variable initially set to null (I called it readWordBuffer). When a word is followed by a punctuation mark, the punctuation mark is placed in readWordBuffer. Just inside the loop in step 2, if readWordBuffer is not null, then I set word equal to readWordBuffer, set readWordBuffer to null and return word.

There are many other ways of dealing with the case of punctuation marks following a word without intervening whitespace: a) You could have readWord() return a String array. b) You could return a word with any punctuation mark at the end. Then, the last character of the returned String can be checked for a punctuation mark. If present, the returned word can be split. c) You can use the BufferedReader’s mark() and reset() methods to push punctuation marks back into the input steam if and only if word already contains alphabetic characters. Then, the next call to readWord() will get and return the punctuation mark.
CS-259 Lab 8: Drawing a String

Before attempting to read the bunch of books on the website, create a 1 line test file with a few special cases and verify it is working correctly. Then create a 3 line file. Try progressively larger files of test cases you hand build until you are fairly sure things are working as they should.

For example, the given data file: CatInTheHat_Line96-102.txt has the text:

So, "said the cat in the hat,
"so
so
so SO...
I will show you another good game that I know!"

Then he ran out.

If read correctly, has the following list of words:

```
word=[So]  word=[so]  word=[that]
word=[,]  word=[.]  word=[I]
word=[said]  word=[.]  word=[know]
word=[the]  word=[.]  word=[.]
word=[cat]  word=[I]  word=[Then]
word=[in]  word=[will]  word=[he]
word=[the]  word=[show]  word=[ran]
word=[hat]  word=[you]  word=[out]
word=[,]  word=[another]  word=[.]
word=[so]  word=[good]  word=[game]
```

Note: The all uppercase word "SO" was not counted as a word, but "I" was counted (since "I" has only one letter, the word can be uppercase and it is not ignored).

Note: The above example just shows each word as it is read. When added to the Markov chain, no word must ever appear with different capitalization and two periods whose parent is also a period would not be added.

Do not put the entire input into a list and then process it with another pass through the data. I know it is quite tempting to many students, but really, we don't need to hold an entire novel in a list. Process the data as read it one word at a time.

You may find some of the methods in java.util.TreeMap and java.util.Collections quite useful.

Have fun!