Programming Exercise: WordGram Class

In this exercise, you will study the WordGram class, a class that will make it easier to extend Markov word prediction. Here, you will add more functionality to the WordGram class and test the new method you add to make sure it works before using this class with MarkovWord. You can get help with some of these methods in the videos.

For this assignment you will get two additional starter files.

- The class **WordGram** is a class that represents words that have an ordering. For example, a WordGram might be the four words “this” “is” “a” test”, in this order. The WordGram class has two private variables, a String array **myWords** to store the words in order, one word per slot, and a private integer **myHash** you will use to be able to use WordGrams as a key with a HashMap. This class has several methods:
    - The constructor has three parameters, a String array named **source**, an integer named **start**, and an integer named **size**. The constructor copies the **size** number of words from **source** starting at the position **start** into a new WordGram.
    - The method **wordAt** has one integer argument name **index**. This method returns the word in the WordGram at position **index**.
- The class **WordGramTester** has methods for testing the WordGram. You may find these helpful in testing the methods you write.
    - The void method **testWordGram** builds and prints several WordGrams from a String.
    - The void method **testWordGramEquals** tests if two WordGrams are equivalent.

Assignment 1: Complete WordGram

In this assignment you will complete the WordGram class that has been started for you. It should have basic methods for **length**, **toString**, **equals**, and **shiftAdd**.

Specifically, for this assignment, you will:

- Write the method **length** that has no parameters and returns the length of the WordGram. This method has been started for you.
- The method **toString** that has no parameters. It prints a WordGram out, showing all the words in the WordGram on one line separated by spaces. This method has been started for you.
- Write the method **equals** that has one parameter of type Object named **o**. This method returns true if two WordGrams are equal and false otherwise. This method has been started for you.
- Write the method **shiftAdd** that has one String parameter **word**. This method should return a new WordGram the same size as the original, consisting of each word shifted down one index (for example the word in slot 1 would move to slot 0, the word in slot 2 would move to slot 1, etc.) and **word** added to the end of the new WordGram. Be sure to test this method. For example, if a WordGram of size 4 is “this” “is” “a” “test”, and **shiftAdd** is called with the argument “yes”, then the method would return a new WordGram ”is” “a” “test” “yes”. This method should not alter the WordGram on which it is called.

It is important that you test your code well before using the WordGram class in the next exercise.

Assignment 2: MarkovWord with WordGram

In this assignment you will create a MarkovWord class that works for any number of words and uses the WordGram class to handle those words.

Specifically, for this assignment, you will:

- Create the **MarkovWord** class that implements **IMarkovModel**. This class should have three private variables, a String array named **myText**, a Random variable named **myRandom**, and an integer variable named **myOrder**. This class should have the following methods, similar to what the MarkovWordOne class had, but extended for handling a larger number of words. You should copy the body of MarkovWordOne and then modify it. The methods in MarkovWord are:
    - A constructor with one integer parameter that is the order (how many words to use in prediction). This method should initialize **myOrder** and **myRandom**.
    - The void method **setRandom** has one integer parameter named **seed**. Using this method will allow you to generate the same random text each time, which will help in testing your program. This method should be the same as it was in MarkovWordOne.
    - The void method **setTraining** has one String parameter named **text**. The String text is split into words and stored in **myText**. The words are used to initialize the training text. This method should be the same as it was in MarkovWordOne.
    - The **indexOf** method has three parameters, a String array of all the words in the training text named **words**, a WordGram named **target**, and an integer named **start** indicating where to start looking for a WordGram match in words. This method should return the first position from **start** that has words in the array **words** that match the WordGram **target**. If there is no such match then return -1.
    - The **getFollows** method has one WordGram parameter named **kGram**. This method returns an ArrayList of all the single words that immediately follow an instance of the WordGram parameter somewhere in the training text. This method should call **indexOf** to find these matches.
    - The **getRandomText** method has one integer parameter named **numWords**. This method generates and returns random text that has **numWords** words. This class generates each word by randomly choosing a word from the training text

that follows the current word in the training text. This method has already been written for you. However, it does not work completely until you have completed the **getFollows** method. You may want to use the **shiftAdd** method you wrote in WordGram.

You should test some of these methods with a small example to make sure they work correctly. Consider printing out the follow set for each WordGram.

For example, if you run MarkovRunner and create a MarkovWord of order 3 with random seed 643, and run it on confucius.txt, the first line of output should be:

```
failure. The sense of his wasted powers may well have tempted
```

Assignment 3: Efficient MarkovWord with WordGram

Again, the **getRandomText** method is inefficient. Suppose it sees the WordGram of order 2 “this” “is” 50 times. Each time, it will calculate the follow set again, which could take a long time if the training text is long. Instead you will create a HashMap of a WordGram to a follow set.

Specifically, you should:

- HashMap does not know how to hash a WordGram. In the WordGram class, you will need to add a method named **hashCode** that has no parameters. This method should return an integer that is a hash code that represents the WordGram. Note that String has a method **hashCode**. You may want to create a String from the WordGram and use the String **hashCode** method.
- Write a new class named **EfficientMarkovWord** (make a copy of MarkovWord to start with) that implements IMarkovModel and that builds a HashMap to calculate the **follows** ArrayList for each possible WordGram only once, and then uses the HashMap to look at the list of characters following when it is needed. This class should include:
    - a method named **buildMap** to build the HashMap (Be sure to handle the case at the end where there is not a follow character. If that WordGram is not in the HashMap yet, then it should be put in mapped to an empty ArrayList. If that key is already in the HashMap, then do not enter anything for this case.)
    - a **getFollows** method, but this **getFollows** method should be much shorter, as it can look up the WordGram, instead of computing it each time.
- To test your HashMap to make sure it is built correctly, write the void method **printHashMapInfo** in the EfficientMarkovWord class. This method should print out the following information about the HashMap:
    - Print the HashMap (all the keys and their corresponding values). Only do this if the HashMap is small.
    - Print the number of keys in the HashMap
    - Print the size of the largest value in the HashMap—that is, the size of the largest ArrayList of characters
    - Print the keys that have the maximum size value.
- Write a new method named **testHashMap** in the MarkovRunner class. This method should create an order-2 EfficientMarkovWord with

    - seed 42
    - the training text is “this is a test yes this is really a test”
    - the size of the text generated is 50
- In the EfficientMarkovWord class, call **printHashMapInfo** after building the HashMap. You should see that the HashMap has the following information:
    - It has 7 keys in the HashMap
    - The maximum number of elements following a key is 2
    - There is one key that has two follow items. The key is “this” “is” and the follow words are “a” and “really”.
- Run another example. This one will test when the WordGram at the end of the training text is unique. In the MarkovRunner class, leave the order at 2, the random seed at 42, and the size of the text generated at 50. Set the training text to the following String:
    - “this is a test yes this is really a test yes a test this is wow”
- In the EfficientMarkovWord class, call printHashMapInfo after building the HashMap. You should see that the HashMap has the following information:
    - It has 10 keys in the HashMap
    - The maximum number of elements following a key is 3
    - There are two WordGrams that each have three follow items. The key “this” “is” has the follow words are “a”, “really” and “wow”. The key “a” “test” has the follow words “yes”, “yes”, and “this”.
    - Note the follow of “is” “wow” should be [ ], an empty array.
- After running it, you’ll probably want to comment out the call to **printHashMapInfo**.
- In the MarkovRunner class, create a void method named **compareMethods** that runs a MarkovWord and an EfficientMarkovWord. In particular,
    - Make both order-2 Markov models
    - Use seed 42 and set the length of text to generate to be 100
    - Both should call **runModel** that generates random text three times for each.

Run the MarkovWord first and then the EfficientMarkovWord with the file “hawthorne.txt” as the training text. One of them should be noticeably faster. You can calculate the time each takes by using **System.nanoTime()** for the current time.