Background (Not necessary to complete assignment)

Markov processes are widely used in Computer Science and in analyzing different forms of data. Part II of this assignment offers an occasionally amusing look at text (it's more fun than counting words) by using a Markov process to generate random text based on a training text. When run in reverse (we won't do that in this assignment), it's possible to identify the source of an unknown text based on frequency of letters and words. This process can be used to identify SPAM or to ascertain if Bacon wrote Romeo and Juliet.

If you're on Facebook, you can use the what-would-i-say FB (or Android) app, described here http://what-would-i-say.com/about.html as "Technically speaking, it trains a **Markov Bot** based on mixture model of **bigram and unigram probabilities** derived from your past post history."

You can also read about the so-called "Infinite Monkey Theorem" via its [Wikipedia entry.](https://en.wikipedia.org/wiki/Infinite_monkey_theorem) This assignment has its roots in several places: a story named *Inflexible Logic* now found in pages

91-98 from [*Fantasia Mathematica* (Google Books)]() and reprinted from a 1940 New Yorker story called by Russell Maloney.
The true mathematical roots are from a 1948 monolog by Claude Shannon, [A Mathematical Theory of Communication]() which discusses in detail the mathematics and intuition behind this assignment. This assignment has its roots in a Nifty Assignment designed by Joe Zachary from U. Utah, assignments from Princeton designed by Kevin Wayne and others, and the work done at Duke starting with Owen Astrachan and continuing with Jeff Forbes, Salman Azhar, and the UTAs from Compsci 201.

Git

Fork, clone, and import the cloned project from the file system. Use this URL from the course GitLab site: https://coursework.cs.duke.edu/201spring19/markov1-wordgram **. *Be sure to fork first*** (see screen shot). Then Clone using the SSH URL after using a terminal window to cd into your Eclipse workspace. Recall that ***you should import using*** the File>Import>General>Existing Projects into Workspace -- then navigate to where you cloned the diyad project.

DO NOT DO NOT import using the Git open -- use General>Existing Projects Into Workspace.

Pushing to Git

When you make a series of changes you want to 'save', you'll push those changes to your GitHub repository. You should do this after major changes, certainly every hour or so of coding. You'll need to use the standard Git sequence to commit and push to GitHub:

```
git add .
git commit -m 'a short description of your commit here'
git push
```

Overview of `WordGram`

Implement a class **`WordGram`** that represents a sequence of words or strings, just like a Java **`String`** represents a sequence of characters. As described below, implement the constructor and all stub-methods so you pass the provided tests and adhere to the design guidelines here.

Just as the Java **`String`** class is immutable, the **`WordGram`** class you implement will be an immutable sequence of strings. Immutable means that once a **`WordGram`** object has been created, it cannot be modified. You cannot change the contents of a **`WordGram`** object, you can create a new **`WordGram`**. String concatenation works similarly.

The number of strings contained in a **`WordGram`** is sometimes called the order of the **`WordGram,`** and we sometimes call the **`WordGram`** an order-k **`WordGram`**, or a k-gram -- the term used in the Markov program you'll implement for Part II. Some examples of order-3 **WordGram** objects include:

"cat"	"sleeping"	"nearby"

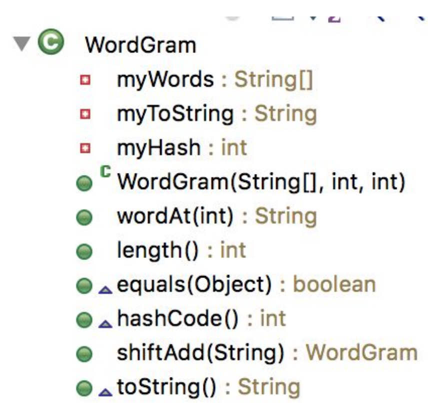
and

"chocolate"	"doughnuts"	"explode"

You'll construct a **WordGram** object by passing an array, a starting index, and the size (or order) of the **WordGram**. You'll ***store the strings in an array instance variable*** by copying them from the array passed to the constructor.

Implementing **WordGram**

You're given an implementation of **WordGram.java** with stub (unimplemented) methods and a stub constructor. See the screenshot from Eclipse to the right that indicates the required methods, constructors, and the three **private** instance variables you'll create. In the **WordGram** class you get in the starter code these methods are not correct, as you can see if you run the JUnit tests in **WordGramTester**. You'll follow these general steps to provide a correct implementation.



1. Replace the stub/incomplete methods in **WordGram** with working versions. In particular, you should implement the following methods and constructor:
    - The constructor **WordGram(String[] words, int index, int size)**
    - **toString()**
    - **hashCode()**
    - **equals(Object other)**
    - **length()**
    - **shiftAdd(String last)**

    For **hashCode**, **equals**, and **toString**, your implementations should conform to the specifications as given in the [documentation for **Object**.]

2. Test these methods using the JUnit tests in **WordGramTester**.

3. Run **`WordGramDriver`** and answer questions in the analysis.txt file in the analysis folder.

`WordGram` Constructors and Methods

(1) Implement the Constructor

The constructor for **`WordGram`**:

```
public WordGram(String[] source, int start, int size)
```

should store **`size`** strings from the array **`source`**, starting at index **`start`** (of **`source`**) into a private **`String`** array instance variable **`myWords`** of the **`WordGram`** class. The array **`myWords`** should contain exactly **`size`** strings.You should include three instance variables in **`WordGram`**:

```
private String[] myWords;
private String myToString;
private int myHash;
```

You should give each of these a value in the constructor, you'll change the values given to **`myToString`** and **`myHash`** in the code you get when you implement the methods **`.toString()`** and **`.hashCode()`**, respectively.

(2) Implement and override method `toString()`

The **`toString()`** method should return a printable **`String`** representing all the strings stored in the **`WordGram`**. This should be a single **`String`** storing each of the values in instance variable **`myWords`** separated by a space. You can do this using the static [**`String.join`**] method with a first parameter of a single-space: **`" "`** and the second parameter the instance variable **`myWords`**.

Don't recompute this String each time `toString` is called -- store the **`String`** in instance variable **`myToString`**. For full credit your code will only call **`String.join`** the first time **`.toString()`** is called and will then use the value stored in **`myToString`** on subsequent calls. This is because once we obtain the **`String`** representation of this **`WordGram`** object, it cannot change (**`WordGram`** is immutable), so there’s no need to recompute it when **`toString()`** is called again. You can test the default value of **`myToString`** to see if you need to assign to it once.

(3) Implement and override method `hashCode()`

The **`hashCode()`** method should return an **`int`** value based on all the strings in instance field **`myWords`**. A simple and efficient way to calculate a hash value is to call **`this.toString()`**

and to use the hash-value of the resultant `String` created and returned by `this.toString()` -- ***you should use this method in calculating hash values for `WordGram` objects.***

Don't recompute the hash value each time `hashCode` is called -- it can't change since `WordGram` objects are immutable. For full credit you'll only call `.toString().hashCode()` the first time `WordGram.hashCode()` is called, your code will store this value in `myHash`, and use the stored value on subsequent calls.

(4) Implement and override method `equals()`

The `equals()` method should return `true` when the parameter passed is a `WordGram` object with ***the same strings in the same order*** as this object. Your code should test the object parameter with the `instanceof` operator to see if the parameter is a `WordGram`. You're given code that makes this test and returns false when the parameter is not a `WordGram` object.

If the parameter is a `WordGram` object, it can be cast to a `WordGram`, e.g., like this:

```
WordGram wg = (WordGram) other;
```

Then the strings in the array `myWords` of `wg` can be compared to this object's strings in `this.myWords`. Note that `WordGram` objects of different lengths cannot be equal.

(5) Implement the method `length()`

The `length()` method should return the order or size of the `WordGram` -- this is the number of words stored in the instance variable `myWords`.

(6) Implement the method `shiftAdd()`

The `shiftAdd()` method should create and return a ***new*** `WordGram` object with k entries (where *k* is the order of this `WordGram`) whose first *k-1* entries are the same as the last *k-1* entries of this `WordGram`, and whose last entry is the parameter `last`.

For example, if `WordGram w` is `{"apple", "pear", "cherry"}` then the method call `w.shiftAdd("lemon")` should return a new `WordGram {"pear", "cherry", "lemon"}`.

The call `w.shiftAdd(string)` is meant to be an analog of the call `s.substring(1).concat(char)` for a String object `s`.

Note: To implement `shiftAdd` you'll need to create a new `WordGram` object, say referenced by a local variable `wg`. You'll be able to assign to the instance variables of this `wg` object since any `WordGram` method can access private state of another `WordGram` object.

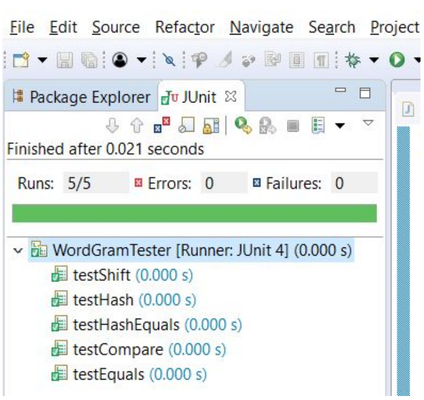
JUnit Tests Explained

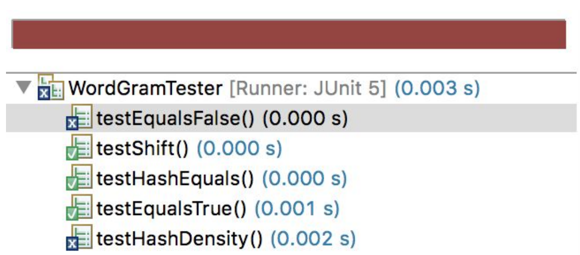
To test your **`WordGram`** class you're given testing code. This code tests individual methods in your class, such tests are called ***unit tests*** you'll need to use the standard JUnit unit-testing library with the WordgramTest.java file to test your implementation. In this assignment you'll be using JUnit 5. This is new as of Spring 2019.

Rather than run as a Java Program, you'll need to choose Run as JUnit test -- first use the "Run As" option in the Run menu as shown on the left below. You have to select the JUnit option as shown on the right. Most of you will have that as the only option.

There are several tests in **`WordGramTester.java`**: **`testEqualsTrue()`**, **`testEqualsFalse()`** which check the correctness of **`.equals`**, **`testHashEquals()`** which checks the consistency of equals and hashing, **`testHashDensity()`** which checks the "performance" of the **`.hashCode`** method, and **`testShift()`** which checks, *to some degree*, the correctness of **`shiftAdd`**.

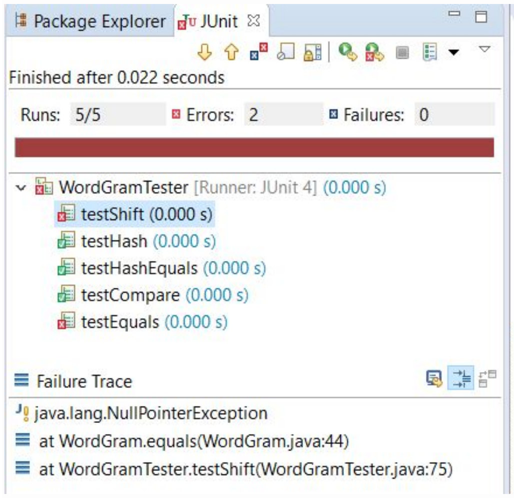
If the JUnit tests pass, you'll get all green as shown on the left below. Otherwise you'll get red — on the right below — and an indication of all the tests that fail. Choose one method to fix as needed and then go on to more tests. The red was obtained from the code you're given. You'll





work to make the testing all green.

Additionally, note the small icon to the left of the test that you failed. If it's blue, it means you are not producing the expected output, as shown above. If it's red, it means your code generated a runtime exception while the JUnit test was running, as shown below:



Submitting

Push your code to Git. Do this often. You can use the autograder on Gradescope to test your code. UTAs will be looking at your source code to view documentation and your analysis.txt file, but you will be able to see the autograding part of your grade -- worth 14 points. Note that you must complete the Reflect form, but you should NOT complete the reflect form until you're done with all the coding portion of the assignment. Since you may uncover bugs from the autograder, you should wait until you've completed debugging and coding before completing the reflect form.

Reflect Form

http://bit.ly/201spring19-wordgram-reflect

Analysis

Answer all the questions in the analysis.txt file in the analysis folder. These questions are reproduced below. To change the size of the WordGram by changing the WordGramDriver static instance variable **`WSIZE`**.

> *YOUR NAME and YOUR NETID*
>
> *Run WordGramDriver for wordgrams of size 2-10 and record the number of WordGram values/objects that occur more than once as reported by the runs. For example, with WSIZE = 2, which generates 2-grams, the output of benchmark and benchmarkShift each indicates that the total # wordgrams generated is 177,634 and that the # unique wordgrams is 117,181*

This means there are 177,634 - 117,181 = 60,453 WordGram values that occur more than once. Find these same values for other orders of k and complete the table below for different k-grams/different values of WSIZE

WSIZE	*# duplicates*
2	*60,453*
3	*10,756*
4	
5	
6	
7	
8	
9	
10	

=====
Explain in your own words the conceptual differences between the benchmark and benchmarkShift methods.
Answer these questions:

(1) Why the results of these methods should be the same in terms of changes made to the HashSet parameter passed to each method.

(2) What are the conceptual differences between the two benchmarking methods

(3) Is the total amount of memory allocated for arrays the same or different in the two methods? Account for arrays created in the methods and arrays created by WordGram objects. Try to be quantitative in answering.

Grading

For this program grading will be:

Points	Grading Criteria
14	correctness of `WordGram` constructor and methods and other results reported by autograder, e.g., API.
6	Answers to analysis questions
2	Comments and style of your code

We will map total points you earn to scores as follows. This means if you lose three points, you receive the same score as losing zero points: an A. We will record the letter grade as your grade for this assignment.

19-22: A
15-18: B
11-14: C
5-10: D