# Programming Exercise: Improving GladLibs

## Assignment 1: Codon Count

Write a program to find out how many times each codon occurs in a strand of DNA based on reading frames. A strand of DNA is made up of the symbols C, G, T, and A. A codon is three consecutive symbols in a strand of DNA such as ATT or TCC. A reading frame is a way of dividing a strand of DNA into consecutive codons. Consider the following strand of DNA = "CGTTCAAGTTCAA".

There are three reading frames.

- The first reading frame starts at position 0 and has the codons: "CGT", "TCA", "AGT" and "TCA". Here TCA occurs twice and the others each occur once.
- The second reading frame starts at position 1 (ignoring the first C character) and has the codons: "GTT", "CAA", "GTT", "CAA". Here both GTT and CAA occur twice.
- The third reading frame starts at position 2 (ignoring the first two characters CG) and has the codons: "TTC", "AAG", "TTC". Here TTC occurs twice and AAG occurs once.

A map of DNA codons to the number times each codon appears in a reading frame would be helpful in solving this problem.

Specifically, you should do the following:

- Create a new class for this problem
- Create a private variable to store a HashMap to map DNA codons to their count.
- Write a constructor to initialize the HashMap variable.
- Write a void method named **buildCodonMap** that has two parameters, an int named **start** and a String named **dna**. This method will build a new map of codons mapped to their counts from the string **dna** with the reading frame with the position **start** (a value of 0, 1, or 2). You will call this method several times, so be sure to make your map is empty before building it.
- Write a method named **getMostCommonCodon** that has no parameters. This method returns a String, the codon in a reading frame that has the largest count. If there are

several such codons, return any one of them. This method assumes the HashMap of codons to counts has already been built.

- Write a void method named **printCodonCounts** that has two int parameters, **start** and **end**. This method prints all the codons in the HashMap along with their counts if their count is between **start** and **end**, inclusive.
- Write a tester method that prompts the user for a file that contains a DNA strand (could be upper or lower case letters in the file, convert them all to uppercase, since case should not matter). Then for each of the three possible reading frames, this method
    - builds a HashMap of codons to their number of occurrences in the DNA strand
    - prints the total number of unique codons in the reading frame
    - prints the most common codon and its count
    - prints the codons and their number of occurrences for those codons whose number of occurrences in this reading frame are between two numbers inclusive

For example, for the string above and here: “CGTTCAAGTTCAA,” also in **smalldna.txt**, if we run our program and print the output requested above and specify to print codons and counts for those codons whose counts are between 1 and 5 inclusive, we might get the output:

```
Reading frame starting with 0 results in 3 unique codons
  and most common codon is TCA with count 2
Counts of codons between 1 and 5 inclusive are:
CGT    1
TCA    2
AGT    1

Reading frame starting with 1 results in 2 unique codons
  and most common codon is CAA with count 2
Counts of codons between 1 and 5 inclusive are:
CAA    2
```

```
GTT    2

Reading frame starting with 2 results in 2 unique codons
  and most common codon is TTC with count 2
Counts of codons between 1 and 5 inclusive are:
TTC    2
AAG    1
```

Note: The convention for text files is to end with a line break, so the above DNA string will have length 13 if directly defined as a String in your tester method, but length 14 if the text file containing it is read in as a FileResource. You can deal with this in several ways. One is to use the String method .trim() (http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#trim() ) to get rid of white space. Another way to guard against identifying the final “AA “ as a codon would be adding a check that requires the last element of each codon be a letter before adding to the codon map.

## Assignment 2: Words in Files

Write a program to determine which words occur in the greatest number of files, and for each word, which files they occur in.

For example, consider you are given the four files: **brief1.txt**, **brief2.txt**, **brief3.txt**, and **brief4.txt**.

**brief1.txt** is:

```
cats are funny and cute
```

**brief2.txt** is:

```
dogs are silly
```

**brief3.txt** is:

```
love animals cats and dogs
```

**brief4.txt** is:

```
love birds and cats
```

The greatest number of files a word appears in is three, and there are two such words: “cats” and “and”.

“cats” appears in the files: **brief1.txt**, **brief3,txt**, **brief4.txt**

“and” appears in the files: **brief1.txt**, **brief3.txt**, **brief4.txt**

To solve this problem, you will create a map of words to the names of files they are in. That is, you will map a String to an ArrayList of Strings. Then you can determine which ArrayList value is the largest (has the most filenames) and its key is thus, a word that is in the most number of files.

Specifically, you should do the following:

- Create a new class called **WordsInFiles**. Put all the remaining listed items in this class.
- Create a private variable to store a HashMap that maps a word to an ArrayList of filenames.
- Write a constructor to initialize the HashMap variable.

- Write a private void method named **addWordsFromFile** that has one parameter **f** of type File. This method should add all the words from **f** into the map. If a word is not in the map, then you must create a new ArrayList of type String with this word, and have the word map to this ArrayList. If a word is already in the map, then add the current filename to its ArrayList, unless the filename is already in the ArrayList.
- Write a void method named **buildWordFileMap** that has no parameters. This method first clears the map, and then uses a DirectoryResource to select a group of files. For each file, it puts all of its words into the map by calling the method **addWordsFromFile**. The remaining methods to write all assume that the HashMap has been built.
- Write the method **maxNumber** that has no parameters. This method returns the maximum number of files any word appears in, considering all words from a group of files. In the example above, there are four files considered. No word appears in all four files. Two words appear in three of the files, so **maxNumber** on those four files would return 3. This method assumes that the HashMap has already been constructed.
- Write the method **wordsInNumFiles** that has one integer parameter called **number**. This method returns an ArrayList of words that appear in exactly **number** files. In the example above, the call `wordsInNumFiles(3)` would return an ArrayList with the words "cats" and "and", and the call `wordsInNumFiles(2)` would return an ArrayList with the words "love", "are", and "dogs", all the words that appear in exactly two files.
- Write the void method **printFilesIn** that has one String parameter named **word**. This method prints the names of the files this word appears in, one filename per line. For example, in the example above, the call `printFilesIn("cats")` would print the three filenames: **brief1.txt**, **brief3.txt**, and **brief4.txt**, each on a separate line.
- Write the void method **tester** that has no parameters. This method should
    - call **buildWordFileMap()** to select a group of files and build a HashMap of words, with each word mapped to an ArrayList of the filenames this word appears in.
    - determine the maximum number of files any word is in, considering all words.
    - determine all the words that are in the maximum number of files and for each such word, print the filenames of the files it is in.

- (optional) If the map is not too big, then you might want to print out the complete map, all the keys, and for each key its ArrayList. This might be helpful to make sure the map was built correctly.

## Assignment 3: Maps Version of GladLibs

Start with your GladLibs program you completed earlier in this lesson. Make a copy of it and call it **GladLibMap.java**. Now modify this program to use one HashMap that maps word types to ArrayList of possible words to select. Your program should still work for the additional categories verbs and fruits and should not use duplicate words from a category. Specifically, you should make the following adjustments to this program:

- Replace the ArrayLists for **adjectiveList**, **nounList**, **colorList**, **countryList**, **nameList**, **animalList**, **timeList**, **verbList**, and **fruitList** with one HashMap **myMap** that maps a String representing a category to an ArrayList of words in that category. Caution: Don't replace the ArrayList representing the words that have already been used!
- Create the new HashMap in the constructors.
- Modify the method **initializeFromSource** to create an Array of categories and then iterate over this Array. For each category, read in the words from the associated file, create an ArrayList of the words (using the method **readIt**), and put the category and ArrayList into the HashMap.
- Modify the method **getSubstitute** to replace all the if statements that use category labels with one call to **randomFrom** that passes the appropriate ArrayList from **myMap**.
- Run your program to make sure it works.
- Write a new method named **totalWordsInMap** with no parameters. This method returns the total number of words in all the ArrayLists in the HashMap. After printing the GladLib, call this method and print out the total number of words that were possible to pick from.
- Write a new method named **totalWordsConsidered** with no parameters. This method returns the total number of words in the ArrayLists of the categories that were used for a particular GladLib. If only noun, color, and adjective were the categories used in a GladLib, then only calculate the sum of all the words in those three ArrayLists. Hint: You will need to keep track of the categories used in solving the GladLib, then compute this total.