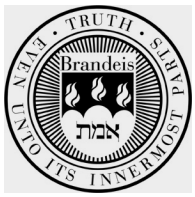

COMPUTER SCIENCE 12B-1 (SPRING TERM, 2017)
PROGRAMMING IN JAVA

PROGRAMMING ASSIGNMENT 2

Due Date: Monday 18 September 2017, 11:55pm

Overview:

This assignment focuses on **arrays** and **classes**. You will implement a data structure (a class) called `LetterInventory` that records the inventory of letters of the alphabet in a given string.

You will need to download the following files from LATTE: `FrequencyAnalysis.java`, `cryptogram.txt`, `Test1.java`, `Test2.java`, `Test3.java`, `text1.txt`, `test2.txt`, `test3.txt`

Translating the Ciphertext

One possible client for this data structure is a class that performs frequency analysis of letters in a cryptogram to decode it. One type of cryptogram which frequency analysis of letters is often very useful for is substitution ciphers. A substitution cipher is a type of cryptogram where all occurrences of each particular letter are all replaced with a single other letter. For instance, if the original text were "*hello i like bananas*", one possible substitution cipher would make the following replacements:

$a \rightarrow v$ $b \rightarrow a$ $e \rightarrow t$ $h \rightarrow x$ $i \rightarrow q$ $k \rightarrow o$ $l \rightarrow p$ $n \rightarrow u$ $o \rightarrow r$ $s \rightarrow w$

Then, the encrypted text would be:

xtppr q pqot avuvuvw.

Submission Overview

We have given you a client implementation of a `FrequencyAnalysis` program which uses your `LetterInventory`. Once your `LetterInventory` class is working, you should be able to run the `FrequencyAnalysis` on `cryptogram.txt` to decode it. The results will be stored in `decodedCryptogram.txt`.

Turn in the following files on LATTE:

- `LetterInventory.java` – A class that keeps track of an inventory of letters of the alphabet in a given string.

- `decodedCryptogram.txt` – The decoded version of the cryptogram, `cryptogram.txt`

You will need the support files `FrequencyAnalysis.java` and `cryptogram.txt` from LATTE to finish decrypting the cryptogram; place these in the same folder as your program or project. The code you submit must work properly with unmodified versions of the supporting files.

Implementation Details

Your `LetterInventory` class should:

- Store the inventory (how many *a*'s, how many *b*'s, etc.) as an array with 26 counters (one for each letter) along with any other data fields you find that you need
- Ignore the case of the letters (e.g., "*a*" and "A")
- Ignore non-alphabetic characters (e.g., digits, punctuation, etc.)
- Introduce a class constant for the value 26 to make the class more readable (so the number 26 should only appear once in your program)
- Should NOT have any extra public methods or have any extra behavior beyond what this spec describes

Constructors

The `LetterInventory` class should have the following two constructors:

- `public` **`LetterInventory`**`():` Constructs an empty inventory (all counts are 0).
- `public` **`LetterInventory`**`(String data):` Constructs an inventory (a count) of the alphabetic letters in data (the given string). Uppercase and lowercase letters should be treated as the same. All non-alphabetic characters should be ignored

Hint: Be careful to remove redundancies between constructors.

Methods

The `LetterInventory` class should have the following public methods:

1. `public int` **`get`**`(char letter):` Returns the number of times the given letter appears in this inventory. Letter can be lowercase or uppercase (your method shouldn't care). If a non-alphabetic character is passed, your method should throw an `IllegalArgumentException`.
2. `public void` **`set`**`(char letter, int value):` Sets the count for the given letter to the given value. Letter can be lowercase or upper- case. If a

non-alphabetic character is passed or if value is negative, your method should throw an `IllegalArgumentException`.

3. `public int` **`size`**`():` Returns the sum of all of the counts in this inventory. This operation should be "fast" in the sense that it should store the size rather than computing it each time the method is called.
4. `public boolean` **`isEmpty`**`():` Returns true if this inventory is empty (all counts are 0). This operation should be "fast" in the sense that it shouldn't loop over the array each time the method is called.
5. `public String` **`toString`**`()` : Returns a String representation of the inventory with all the letters in lowercase, in sorted order, and surrounded by square brackets. The number of occurrences of each letter should match its count in the inventory. For example, an inventory of 4 *a*'s, 1 *b*, 1 *g* and 1 *m* would be represented as "[*aaaabgm*]" .
6. `public LetterInventory` **`add`**`(LetterInventory other):` Constructs and returns a **new** `LetterInventory` object that represents the sum of this `LetterInventory` and the other given `LetterInventory`. The counts for each letter should be added together. The two `LetterInventory` objects being added together (this and other) should not be changed by this method. You might be tempted to implement the add method by calling the `toString` method, but you may not use that approach, because it would be inefficient for inventories with large character counts.

   Below is an example of how the add method might be called:

```
LetterInventory inventory1 = new LetterInventory("George W. Bush");

LetterInventory inventory2 = new LetterInventory("Hillary Clinton");

LetterInventory sum = inventory1.add(inventory2);
```

   The first inventory would correspond to [*beegghorsuw*], the second would correspond to [*achiilllnnorty*] and the third would correspond to [*abceegghhiilllnnoorrstuwy*].

7. `public LetterInventory` **`subtract`**`(LetterInventory other)` Constructs and returns a new `LetterInventory` object that represents the difference of this letter inventory and the other given `LetterInventory`. The counts from the other inventory should be subtracted from the counts of this one. The two `LetterInventory` objects being subtracted (this and other) should not be changed by this method. If any resulting count would be negative, your method should return null.

8. `public double` **`getLetterPercentage`**`(char letter)`
   Returns a double between 0.0 and 1.0 representing the percentage of letters in the inventory that are the given letter. If there are no letters in the inventory, this method should always return 0. If all the letters in the inventory are the given letter, the method should return 1.0 to represent 100 percent. If a non-alphabetic character is passed, your method should throw an `IllegalArgumentException`.

NOTE: You can add more instance method if see fit but you cannot add any static methods in your `LetterInventory` class.

Useful Properties of Strings and Characters

You will need to know certain things about the properties of letters and type char. One of the most important ideas is that the values of type char have corresponding integer values. There is a character with value 0, a character with value 1, a character with value 2 and so on. You can compare different values of type char using less-than and greater-than tests. For example:

```
if(ch>='a'){   ... }
```

All of the lowercase letters appear grouped together in type char (‘a’ is followed by ‘b’ followed by ‘c’, and so on), and all of the uppercase letters appear grouped together in type char (‘A’ followed by ‘B’ followed by ‘C’ and so on). Because of this, you can compute a letter’s displacement (or distance) from the letter ‘a’ with an expression like the following (this expression assumes the variable letter is of type char and stores a lowercase letter):

```
letter - 'a'
```

Use this to find indices and offsets.

Going in the other direction, if you know a char’s integer equivalent, you can cast the result to char to get the character. For example, suppose that you want to get the letter that is 8 away from ‘a’. You could say:

```
char result = (char) ('a' + 8);
```

This assigns the variable result the value ‘i’.

As in these examples, you should write your code for `LetterInventory` in terms of displacement from a fixed letter like ‘a’ rather than including the specific integer value of a character.

You probably want to look at the `String` and `Character` classes for useful methods

(e.g., there is a `toLowerCase` method in each). You will have to pay attention to whether a method is `static` or not. The `String` methods are mostly instance methods, because Strings are objects. The `Character` methods are all static, because `char` is a primitive type. For example, assuming you have a variable called `s` that is a `String`, you can turn it into lowercase by saying:

```
s = s.toLowerCase();
```

This is a call on an instance method where you put the name of the object first. But `chars` are not objects and the `toLowerCase` method in the `Character` class is a `static` method. So, assuming you have a variable called `ch` that is of type `char`, you'd turn it to lowercase by saying:

```
ch = Character.toLowerCase(ch);
```

Development Strategy

The best way to write code is in stages. If you attempt to write everything at once, it will be significantly more difficult to debug, because any bugs are likely not isolated to a single place.

For this assignment we will provide you with a development strategy and some testing code. We provide `Test1.java, Test2.java, Test3.java`. You should run them to test the correctness of your program. These tests are **not** exhaustive; they are meant as an example of what kind of tests we expect you to conduct.

We suggest that you develop the program in the follow four stages:

1. In this stage, we want to test constructing a `LetterInventory` and examining its contents. So, the methods we will implement are the constructors, the `size` method, the `isEmpty` method, the `get` method, and the `toString` method. Even within this stage you can develop the methods slowly. You should implement these methods in the order above. The testing program (`Test1.java`) will test them in this order; so, it will be possible to implement them one at a time.
2. In this stage, we want to add the `set` method which allows the client to change the number of occurrences of an individual letter. The testing program (`Test2.java`) will verify that other methods work properly in conjunction with `set` (the `get`, `isEmpty`, `size`, and `toString` methods).
3. In this stage, we want to include the `add` and `subtract` methods. You should write the `add` method first and make sure it works. The testing program (`Test3.java`) first tests `add`; so, don't worry that the fact that the tests on `subtract` fail initially.
4. Finally, we want to include the `getLetterPercentage` method. A reasonable test of this step is to run the `FrequencyAnalysis` and check that the text makes sense. There is no test class for this stage

Style Guidelines

Unless otherwise specified, your solution should use only material covered so far.

No Extra Data Structures

Your letter inventory should maintain its list of letters internally in a field of type array as specified here. Do not create any other data structures or any other classes from Java's collection framework.

Data Fields

Properly encapsulate your objects by making data your fields private. Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place. Fields should always be initialized inside a constructor or method, never at declaration.

Java Style Guidelines

Appropriately use control structures like loops and if/else statements. Avoid redundancy using techniques such as methods, loops, and factoring common code out of if/else statements. Properly use indentation, good variable names, and types. Do not have any lines of code longer than 80 characters.

Commenting

You should comment your code with a heading at the top of your class with your name, section, and a description of the overall program. All method headers should be commented as well as all complex sections of code. Comments should explain each method's behavior, parameters, return values, and assumptions made by your code, as appropriate.

Grading:

You will be graded on:

- **External Correctness:** The output of your program should match exactly what is expected. Programs that do not compile will not receive points for external correctness.
- **Internal Correctness:** Your source code should follow the stylistic guidelines shown in class. Also, remember to include the comment header at the beginning of your program.
- **One-on-one interactive grading:** By the end of the day that the assignment is due, please make an appointment with your TA for an interactive 10-15 minute grading session. You will receive an email notifying you of the name of the TA who has been assigned to you for this assignment with further instructions on setting up the appointment. (You will be meeting with a different TA for each assignment). One-on-one interactive grading will help you improve your programming skills and avoid repeating mistakes from one assignment to the next.