CSE 143: Computer Programming II — Winter 2021

Take-hom Assessment 1: Letter Inventory *due January 14, 2021 11:59pm*

This assignment will assess your mastery of the following objectives:

- Implement a well-designed Java class to meet a given specification.
- Maintain proper abstraction between the client and implementation of a class.
- Follow prescribed conventions for code quality, documentation, and readability.

Overview

In this assessment, you will implement a class called `LetterInventory` that can be used to keep track of an inventory of letters of the English alphabet. The constructor for the class will take a `String` as a parameter and compute how many of each letter are in that `String` (i.e. how many a's, how many b's, etc.). `LetterInventory` ignores any character that is not an English letter (such as punctuation or digits) and treats upper- and lowercase letters as the same.

Your `LetterInventory` class should include the following constructor:

`public` **`LetterInventory`**`(String data)`
Constructs an inventory (a count) of the alphabetic letters in the given string, ignoring the case of letters and ignoring any non-alphabetic characters.

Your class should also include the following public methods:

You must include *exactly* these method headers—do not add or remove parameters.

`public int` **`get`**`(char letter)`
Returns a count of how many of this letter (case-insensitive) are in the inventory. If a nonalphabetic character is passed, your method should throw an `IllegalArgumentException`.

`public void` **`set`**`(char letter, int value)`
Sets the count for the given letter (case-insensitive) to the given value. If a nonalphabetic character is passed or if `value` is negative, your method should throw an `IllegalArgumentException`.

`public int` **`size`**`()`
Returns the sum of all of the counts in this inventory. This operation should be "fast" in that it should store the size rather than having to compute it each time this method is called.

`public boolean` **`isEmpty`**`()`
Returns `true` if this inventory is empty (i.e. all counts are 0). This operation should be "fast" in that it should not need to examine each of the 26 counts when it is called.

`public String` **`toString`**`()`
Returns a string representation of the inventory with the letters all in lowercase and 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 as, 1 b, 1 l and 1 m would be represented as `"[aaaablm]"`.

`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.

`public LetterInventory` **`subtract`**`(LetterInventory other)`
Constructs and returns a new `LetterInventory` object that represents the result of subtracting the other inventory from this inventory (i.e. subtracting the counts in the other inventory from this objects counts). If any resulting count would be negative, this method should return `null`. The two `LetterInventory` objects being subtracted (`this` and `other`) should not be changed by this method.

Make sure any helper methods are declared `private`.

You may also include any additional private helper methods you think will be helpful.

As an example, the add method could be called as follows:

```
LetterInventory inventory1 = new LetterInventory("Sherlock Holmes");
LetterInventory inventory2 = new LetterInventory("Dr. John Watson");
LetterInventory sum = inventory1.add(inventory2);
```

Here, `inventory1` would contain `[ceehhkllmoorss]`, `inventory2` would contain `[adhjnnoorstw]`, and `sum` would contain `[acdeehhhjkllmnnoooorrssstw]`.

Implementation Guidelines

You should implement this class with an array of 26 counters (one for each letter) along with any other data fields you find that you need. Remember, though, that we want to minimize the number of data fields when possible.

Your class should avoid unnecessary inefficiencies. For example, you might be tempted to implement the `add` method by calling the `toString` method or otherwise building a `String` to pass to the `LetterInventory` constructor. But this approach would be inefficient for inventories with large character counts.

You should introduce a class constant for the value 26 to improve readability.

Character operations

It will be helpful to understand certain deatils of the `char` datatype for this assessment. Many of these details are explained in section 4.3 of the textbook.

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, as in:

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

All of the lowercase letters appear grouped together in type `char` (i.e. `'a'` is followed by `'b'` followed by `'c'`, and so on). All of the uppercase letters appear grouped together similarly. 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'
```

Going in the other direction, if you know a characters 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 do this as follows:

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

This would assign the variable `result` the value `'i'`. As in these examples, you should write your code in terms of displacement from a fixed letter like `'a'` rather than finding and including the specific integer value (e.g. 97) of a character like `'a'`.

Hints

Thought it may not seem like it, the `ArrayIntList` example from lecture provides a good model to use for implementing `LetterInventory`. Pay particular attention to the use of fields, avoiding reimplementation of common functionality, throwing exceptions in error conditions, and documentation/comments.

String and Character

You will likely want to look at the Java `String` and `Character` classes for useful methods. (For example, there is a `toLowerCase` method in each.) You will have to pay attention to whether each 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, if you have a variable called `s` that is a `String`, you can turn it to all lowercase as follows:

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

This is a call to an instance method on an object, so you put the name of the object variable before the dot. But `char` values are *not* objects and the `toLowerCase` method in the `Character` class is a static method. So if you have a variable called `ch` that is of type `char`, you would turn it to all lowercase as follows:

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

Development Strategy

One of the most important techniques for programmers is to develop code in stages rather than trying to write it all at once. (The technical term for this is "iterative enhancement" or "stepwise refinement.") It is also important to be able to test the correctness of your solution at each different stage.

We suggest that you work on your assessment in three stages:

(a) First, work on constructing a `LetterInventory` and examining its contents. We will implement the constructor, the `size` method, the `isEmpty` method, the `get` method, and the `toString` method. Even within this stage, you should develop the methods slowly. First work on the constructor and `size` methods. Then add the `isEmpty` method, then the `get` method, then the `toString` method.

(b) Next, add the `set` method to the class that allows the client to change the number of occurrences of an individual letter.

(c) Finally, include the `add` and `subtract` methods. We recommend writing the `add` method first and making sure it works, then moving on to the `subtract` method.

Code Quality Guidelines

In addition to producing the desired behavior, your code should be well-written and meet all expectations described in the grading guidelines, Code Quality Guide, and Commenting Guide. For this assessment, pay particular attention to the following elements:

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.

Exceptions

The specified exceptions must be thrown correctly in the specified cases. Exceptions should be thrown as soon as possible, and no unnecessary work should be done when an exception is thrown. Exceptions should be documented in comments, including the type of exception thrown and under what conditions.

Commenting

Each method should have a header comment including all necessary information as described in the Commenting Guide. Comments should be written in your own words (i.e. not copied and pasted from this spec) and should not include implemenation details.

Running and Submitting

If you believe your behavior is correct, you can submit your work by clicking the "Mark" button in the Ed assessment. You will see the results of some automated tests along with tentative grades. **These grades are not final until you have received feedback from your TA.**

You may submit your work as often as you like until the deadline; we will always grade your most recent submission. Note the due date and time carefully—**work submitted after the due time will not be accepted**.

Getting Help

If you find you are struggling with this assessment, make use of all the course resources that are available to you, such as:

- Reviewing relevant examples from class
- Reading the textbook
- Visiting office hours
- Posting a question on the message board

Collaboration Policy

Remember that, while you are encouraged to use all resources at your disposal, including your classmates, **all work you submit must be entirely your own**. In particular, you should **NEVER** look at a solution to this assessment from another source (a classmate, a former student, an online repository, etc.). Please review the full policy in the syllabus for more details and ask the course staff if you are unclear on whether or not a resource is OK to use.

Reflection

In addition to your code, you must submit answers to short reflection questions. These questions will help you think about what you learned, what you struggled with, and how you can improve next time. The questions are given in the file `LetterInventoryReflection.txt` in the Ed assessment; type your responses directly into that file.