CSE 143 Winter 2015
Programming Assignment #1: LetterInventory
Due Thursday, January 15, 2014, 11:30 PM

This assignment focuses on arrays, classes, and `ArrayLists`. Turn in the following files using the link on the homework section of the course website:

- `LetterInventory.java` – A class that keeps track of an inventory of letters of the alphabet.
- `decodedCryptogram.txt` – The decoded version of the cryptogram, `cryptogram.txt`, which you will be able to create using your program. (See later pages.)

You will need the support files `FrequencyAnalysis.java` and `cryptogram.txt` from the homework section of the course web site to decrypt the cryptogram; place these in the same folder as your program or project. You should not modify the provided files. The code you submit must work properly with their unmodified versions.

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. Remember, though, we want to minimize the number of data fields whenever possible. Your class should ignore the case of the letters (e.g., "a" and "A" are the same) and ignore anything that is not an alphabetic character (e.g., it ignores punctuation characters and digits). You should introduce a class constant for the value 26 to add to readability.

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 (for example, 'a' and 'A' are the same. All non-alphabetic characters should be ignored.

Methods

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

`public int` **`get`**`(char letter)`
Returns a count of how many of this `letter`s are in the 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`.

`public void` **`set`**`(char letter, int value)`
Sets the count for the given `letter` to the given `value`. `letter` might be lowercase or uppercase. If a non-alphabetic 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 the sense that it should store the size rather than computing it each time the method is called.

`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.

`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 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.

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]`

```
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`.

```
public double getLetterPercentage(char letter)
```

Returns a double representing the percentage of letters in the inventory that are `letter`.
If there are no letters in the inventory, you should return 0.
If a non-alphabetic character is passed, your method should throw an `IllegalArgumentException`.

Useful Properties of Strings and Characters

You will need to know certain things about the properties of letters and type `char`. It might help to look at the section in Chapter 4 of the textbook about the 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'
```

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 like 'a'.

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 `char` values 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);
```

You can read about `String` operations on pages 160 – 166 of the textbook.

Translating the Ciphertext

In this assignment, you've implemented the `LetterInventory` data structure. In particular, you've seen the data structure from the *implementor's view*. 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 *transposition ciphers*. A *transposition 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 transposition cipher would make the following replacements: a → v, b → a, e → t, h → x, i → q, k → o, l → p, n → u, o → r, s → w. Then, the encrypted text would be: `xtppr q pqot avuvuvuw`.

We have given you a *client* implementation of a `FrequencyAnalysis` program, which uses your `LetterInventory`. When your `LetterInventory` is working, you should be able to run the `FrequencyAnalysis` on `ciphertext.txt` to decode it. Turn in the result as `decodedCryptogram.txt`

Development Strategy

One of the most important techniques for software professionals is to develop code in stages rather than trying to write it all at once (the technical term 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 have noticed that many 143 students do not develop their code in stages and do not have a good idea of how to test their solutions. As a result, for this assignment we will provide you with a development strategy and some testing code. We aren't going to provide exhaustive testing code, but we'll give you some good examples of the kind of testing code we want you to write.

We suggest that you develop the program in 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. First, do the constructor and `size` methods. Then, add the `isEmpty` method. Then, add the `get` method. Then, add the `toString` method. The testing program 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 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 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 partial test of this method is to run the `FrequencyAnalysis` and check that the text makes sense.

We will be providing **some** testing code for the first three stages (but not the fourth). You may discuss how to write testing code with other students. Keep in mind that the tests are not guaranteed to be exhaustive. They are meant to be examples of the kinds of tests you should perform.

Style Guidelines and Grading:

A major focus of our style grading is **redundancy**. As much as possible, avoid redundancy and repeated logic in your code. One powerful way to avoid redundancy is to create "helper" method(s) to capture repeated code. It is legal to have additional methods in your `LetterInventory` class beyond those specified here. For example, you may find that multiple methods in your class do similar things. If so, you should create helper method(s) to capture the common code. (You should declare such methods to be `private` rather than `public`, so that outside code cannot call them.)

Your letter inventory should maintain its list of letters internally in a field of type `array` as stated previously. You should not use any other data structures.

Properly **encapsulate** your objects by making any data fields in your class `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.

You should follow good general Java style guidelines such as: appropriately using control structures like loops and `if/else` statements; avoiding redundancy using techniques such as methods, loops, and factoring common code out of `if/else` statements; properly using indentation, good variable names, and types; and not having any lines of code longer than 100 characters in length. (If you have any such lines, split them into two or more lines using a line break.)

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. Also place a comment heading on top of each method, and a comment on any complex sections of your code. Comment headings should use descriptive complete sentences and should be written *in your own words*, explaining each method's behavior, parameters, return values, and assumptions made by your code, as appropriate. The `ArrayIntList` class from lecture provides a good example of the kind of documentation we expect you to include. You do not have to use the pre/post format, but you must include the equivalent information, including exactly what type of exception is thrown if a precondition is violated.

Unless otherwise specified, your solution should use only material taught in class and in the book chapters covered so far.