CSE143—Computer Programming II
Programming Assignment #1
due: Thursday, 4/8/21, 11 pm

In this programming assignment you will practice using arrays and classes. You are to implement a class called LetterInventory that can be used to keep track of an inventory of letters of the alphabet. The constructor for the class takes a String and computes how many of each letter are in the String. This is the information the object keeps track of (how many a's, how many b's, etc). It ignores the case of the letters and ignores anything that is not an alphabetic character (e.g., it ignores punctuation characters, digits and anything else that is not a letter).

Your class should have the following public methods.

Method	Description
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.
int get(char letter)	Returns a count of how many of this letter are in the inventory. Letter might be lowercase or uppercase (your method shouldn't care). If a nonalphabetic character is passed, your method should throw an IllegalArgumentException.
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 nonalphabetic character is passed or if value is negative, your method should throw an IllegalArgumentException
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.
boolean isEmpty()	Returns true if this inventory is empty (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.
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 a's, 1 b, 1 l and 1 m would be represented as "[aaaablm]".
LetterInventory add(LetterInventory other)	Constructs and returns a new LetterInventory object that represents the sum of this letter inventory 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
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 object's counts). If any resulting count would be negative, your method should return null. The two LetterInventory objects being subtracted (this and other) should not be changed by this method

Below is an example of how the add method would 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].

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. You might be tempted to implement the add method by calling the toString method but you are not allowed to use that approach because it would be inefficient for inventories with large character counts. You should introduce a class constant for the value 26 to add to readability.

You will need to know certain things about the properties of letters and type char. There is a section about type char in chapter 4 of the textbook. 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, as in:

```
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 character'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 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 to 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 166—173 of the textbook.

In terms of correctness, your class must provide all of the functionality described above and must satisfy all of the constraints mentioned in this writeup. In terms of style, we will be grading on your use of comments, good variable names, consistent indentation, minimal data fields and good coding style to implement these operations.

The ArrayIntList class discussed in 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. Remember to mention all important behavior that a client would want to know about.

You should name your file LetterInventory.java and you should turn it in electronically from the "Homework" tab on the class web page.

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 are suggesting that you develop the program in three stages:

1. In this stage we want to test constructing a LetterInventory and examining it's contents. So the methods we will implement are the constructor, 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. It this stage we want to add the set method to the class that 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.

We will be providing testing code for each of these three stages and for this program only you are allowed to 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.