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.

Constructors
The LetterInventory class should have the following constructor:

```java
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.
```

Methods
The LetterInventory class should have the following public methods:

```java
public int get(char letter)
Returns a count of how many of the given letter (case-insensitive) are in this inventory. 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 (case-insensitive) to the given value. 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 "[aaaablml]"

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.

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.

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

LetterInventory inventory1 = new LetterInventory("Hello world!");
LetterInventory inventory2 = new LetterInventory("owl");
LetterInventory difference = inventory1.subtract(inventory2);

The first inventory would correspond to [dehllloorw], the second would correspond to [low] and the third would correspond to [dehllor]. If instead we called inventory2.subtract(inventory1), it would return null.

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 details 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:

```java
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):

```java
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:

```java
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.

Hints

Thought it may not seem like it, the ArrayIntList example from class 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 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:

```java
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:

```java
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. The technical term for this development style 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:

1. 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.

2. Next, add the set method to the class that allows the client to change the number of occurrences of an individual letter.

3. 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 and Grading**

In addition to producing the desired behavior, your code should be well-written and meet all expectations described in the General Style Deductions, Style 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 implementation details.

**Testing and Submitting**

While developing your code, you should work locally on your computer in jGRASP. To test your code, you should run the provided Test.java files and copy the printed logs into the Output Comparison Tool to check against the expected output.

If you believe your behavior is correct, you can submit your work by uploading your files and 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 should treat Ed as a final check for your correctness, not as a debugging tool. If you aren’t passing the tests, you should go back to jGRASP to debug your work.

You may submit your work as often as you like until the deadline; we will always grade your most recent
submission. If you submit a version that you later decide you do not want to have graded, you must warn your TA not to grade that version and to wait for a later submission from you.

**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 the IPL or Taylor’s office hours
- Posting a question on the Ed discussion 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 on what types of collaboration are encouraged. Please ask the course staff if you are unclear on whether or not a resource is OK to use.