Summary: Write Java code to generate a camel case version of an input string. Unlike the code in lecture this code will have to handle arbitrary input strings so you will also need to handle error conditions when the input cannot be turned into camel case versions of the input.We will provide a function interface for your code, but the design of the implementation is up to you. Try to design your code to be as clean and readable as possible, using the ideas in Chapters 1, 2, and 3 of the book to inform your implementation.

Background: Camel case (`https://en.wikipedia.org/wiki/Camel_case`) is often used in variable naming. It provides a standard way to condense multi-word ideas into a single name. You will be using it often in this class since it is part of the naming scheme in the style guide we will be using for Java this semester.

Specification: In this assignment you will be writing a single function in the class `CamelCase` with the following signature.

```
public static String camelCase(String inputString)
```

This function will take the input string and if possible produce a lower camel cased version of that string. For a string to be able to be camel cased it must only include whitespace, letters, and digits according to the functions in the Java Character class. If the input string contains other characters you should throw an exception of type `IllegalArgumentExceptions` with an "Invalid Character" error message. If the input string contains words that start with digits you should throw an exception of type `IllegalArgumentExceptions` with a "Invalid Format" error message. If a `null` is passed in you should throw an exception of type `IllegalArgumentException` with a "Null Input" error message. To support good programing style there is an included class `ErrorConstants` that should be used for the error strings for these exceptions. In all other cases the lower camel case version of the string should be returned.

The lower camel case version of the string is defined as follows.

- Divide the string into words, splitting on whitespace.
- Lowercase everything, then uppercase only the first character of each word except the first.
- Join all the words into a single string.

Following is a simple example.

Input:

```
"This is my string"
```

Output:

```
"thisIsMyString"
```

Your repository: (Be sure to have followed the directions in Piazza for setting up your GitHub user-name and password in IntelliJ. Do that first.)

Use the following link to create your own copy of the CamelCaser repository on GitHub:

```
https://classroom.github.com/a/ZqnwYxwE
```

In IntelliJ, create a new project using the following menu command:

```
File -> New -> Project From Version Control -> GitHub
```

A dialog should pop up. Select the appropriate repo from the list labeled "Git Repository URL". Click the "clone" button. Since the version of Java you are using might be different than the one I used to create the project, you might need change the **Project SDK** using the menu command below. Select a Project SDK that you have installed that is Java 1.7 or greater. For some of you a pop-up might appear asking you to choose a different Project SDK.

```
File -> Project Structure
```

To commit your work, use the following command. Write a useful commit message, and be sure to push your work to GitHub.

```
VCS -> Commit Changes
```

Be sure to frequently commit your changes. When writing code, you should break your work into lots of little pieces (15-60 minutes of effort). Your work cycle should be to: 1) pick the next piece that you can implement, 2) implement it, 3) debug it until you are confident that it works, and 4) commit and push it. That way, if anything happens, you never lose more than a small amount of work. **Part of your grade on code review assignments will be based on progressively committing your code.** Note: if your code is broken and not compiling, you probably want to get it working before committing.

Testing: The primary motivation for this assignment is to give you a moderately complicated function for which to write black box tests. Following a test-first philosophy, we're going to ask you to write your tests before writing your code. More precisely, your commits should show some tests written before you finish your implementation.

Your tests should be implemented in the provided `CamelCaserTest` class using Junit4. Each assert should be in a separate test. Name your tests with meaningful names. For most tests, no comments are necessary, because the name of the test is sufficiently explanatory. Write enough tests to give yourself confidence that your implementation will be correct if it passes all of the tests. Exhaustive testing is not possible given that the number of possible Java Strings is bounded only by memory.

To effectively test your code using a much smaller number of tests, use the “Bag of Tricks” we discussed in Lecture 2 (e.g., equivalence classes, boundary conditions, classes of bad data) to pick a useful set of inputs to test. Take an adversarial approach to writing your tests, trying to identify the problematic inputs that are likely to break your implementation and that of other students.

Design and Style:

we will be using the Google Java Style Guide for the code that we write in CS 126. For this assignment, please read the portion of the style guide associated with naming (see the URL below) and follow the conventions described.

`https://google.github.io/styleguide/javaguide.html#s5-naming`